Let's C

C Compiler

Professional Development Tool
for the IBM PC and Compatibles

Mark Williams Company

V4.06 - rec'd 8/17/87

Mark Williams Company — Let's C Registration Form

Please complete this form and return it within 30 days to qualify for maintenance and product updates. This information will help us serve your needs better. Please read and complete all the questions: be specific and check (✓) all items that apply. Your assistance is greatly appreciated.

Customer Data

Name & Title: _____

Company Name: _____

Address: _____

City: _____ State or Country: _____ Zip Code: _____

Daytime Telephone: _____ Extension: _____

Purchased From: _____ Purchase Date: _____

1. How did you hear about this product?
- ☐ Advertisement (magazine): _____
- ☐ Review (magazine): _____
- ☐ Other _____

2. Computer model:

_____

3. Amount of RAM in your computer:
- ☐ 256k ☐ 512k ☐ 640k
- ☐ 1MB ☐ Extended Memory

4. # of floppy disk drives:
- ☐ 1 ☐ 2

5. Do you have a hard disk drive:
- ☐ Yes ☐ No

6. I read the following regularly:
- ☐ Byte ☐ PC Magazine
- ☐ Computer Language ☐ PC Tech Journal
- ☐ Computerworld ☐ PC Week
- ☐ Dr. Dobb's Journal ☐ PC World
- ☐ InfoWorld ☐ Personal Computing
- ☐ Analog ☐ ST Log
- ☐ Antic ☐ START
- ☐ Compute ST ☐ ST Applications
- ☐ Other _____
- ☐ Other _____

7. My level of C programming experience is:
- ☐ beginner ☐ intermediate ☐ professional
- ☐ new to C but professional in other language

8. Describe your current C programming projects:

_____

_____

_____

_____

_____

9. I prefer to purchase software from:
- ☐ retail (full-service) ☐ mail-order
- ☐ retail (discount) ☐ direct from company
- ☐ Other _____

10. This software was purchased for:
- ☐ personal use ☐ business use
- ☐ educational program (please specify)

_____

_____

11. Other software purchase interests (please ✓ items and also elaborate in space provided):
- ☐ Source Level Debugger ☐ C Interpreter
- ☐ C library toolboxes ☐ C tutorial
- ☐ C run-time source code ☐ GEM libraries
- ☐ Cross development tools ☐ Other compilers
- ☐ Other programming tools ☐ Other utilities
- ☐ Embedded applications toolbox

_____

_____

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS • PERMIT NO. 10820 • CHICAGO, IL

POSTAGE WILL BE PAID BY ADDRESSEE

Mark Williams Company
1430 Wrightwood Avenue
Chicago, Illinois 60614

Let's C

C Compiler

Professional Development Tool
for the IBM PC and Compatibles

Mark Williams Company

Revision 3 Printing 5 4 3 2 1

Published by Mark Williams Company, 1430 W. Wrightwood Avenue, Chicago, Illinois 60614.

I created this PDF version of the *csd C Source Debugger* manual in October 2020 using my 1987 hard copy and archived Mark Williams Company documentation sources. The pages at the beginning and end (cover, Registration Form, title page, User Reaction Report, Other Products, Order Form, Software License Agreement, back cover) are scans of my hard copy. The remaining sections were regenerated from the archived sources using the COHERENT version of **troff**. The reconstructed manual has not been carefully proofread.

Stephen Ness
10/19/2020

Table of Contents

CONTENTS

CONTENTS

CONTENTS

CONTENTS

Introduction

Congratulations on choosing **Let's C**, the Mark Williams C compiler for the IBM PC and compatibles. **Let's C** has the state-of-the-art power and flexibility that the professional programmer needs, but is easy enough for the beginner to learn quickly.

Let's C is part of the Mark Williams Company family of C compilers, which supports many different operating systems and processors. The operating systems supported include:

COHERENT	MS-DOS	TOS
CP/M-68K	RMX	VAX/VMS
ISIS-II		

The processors supported include:

PDP-11	68000	80186
Z8001	68020	80286
Z8002	8086	

What is Let's C?

Let's C is a professional C programming system designed for the IBM PC and compatibles. It consists of the following:

- The Mark Williams C compiler, plus an assembler, a preprocessor, and other tools.
- A set of commands selected from the COHERENT operating system, including the MicroEMACS screen editor and the **make** programming discipline.
- A full set of C libraries.
- A set of sample programs, including full source code for the MicroEMACS editor.
- The Mark Williams shell MWS. MWS will help you build commands for **Let's C**, and will accelerate the operation of your software. By using MWS's display interface, you build commands for **Let's C** and its utilities. MWS also includes an accelerator that speeds up **Let's C**. If you prefer to type commands directly into MS-DOS, MWS will let you and it will still accelerate your software for you.

Hardware requirements

Let's C runs on an IBM PC, XT, or AT, or any compatible computer that has at least 320 kilobytes of RAM and either two double-sided floppy disk drives or at least one floppy disk drive and a hard disk.

Changes in release 4.0

Let's C version 4.0 has been greatly expanded and improved over earlier releases. Its new features include the following:

- **Let's C** now compiles into MS-DOS format, rather than the proprietary Mark Williams format used in earlier versions.
- It supports LARGE model as well as SMALL model.
- **Let's C** supports the i8087 mathematics co-processor, to speed up mathematics routines.

- **Let's C** supports **csd**, the revolutionary Mark Williams C source debugger. **csd** can now debug programs in MS-DOS object format, and that are compiled into either SMALL or LARGE model.
- **Let's C** libraries will sense the presence of the i8087 co-processor. If one is present, then it is used to execute mathematics routines; if one is not present, mathematics routines are emulated in software. Your programs will now make maximum use of your computer, whether an i8087 is present or not.
- The **make** programming discipline is included. This helps you to construct large programs that use many modules, with a minimum of difficulty.
- **Let's C**'s assembler, **as**, has been improved to support both LARGE and SMALL model, as well as the i8087 co-processor. It generates MS-DOS object format rather than the Mark Williams proprietary format, as before. **as** now supports a macro processing feature, which allows you to write model-independent versions of your assembly-language programs.
- The utility **fixobj** lets you edit object modules and libraries so they can be linked with object modules generated by **Let's C**. You can now use libraries from any other C compiler with **Let's C**.
- The Mark Williams shell MWS now makes it easy to build commands for **Let's C** and its utilities. MWS also supports RAM compiling, to speed up compilation on your system.
- The MicroEMACS screen editor is now integrated with **Let's C**. If you wish, you can have MicroEMACS display your source code automatically whenever an error occurs during compilation; you can then fix your error and recompile by using only a few keystrokes.
- Floating-point numbers now use IEEE format. Floating-point routines have been rewritten to execute more quickly than before.
- Division of **long**s has been rewritten, and is much faster than in previous versions.
- **Let's C** will now compile programs using i80286 instructions. Although such programs cannot be run on a computer that uses the i8086 microprocessor, they will run more quickly on the IBM PC-AT and compatibles.
- Finally, the manual for **Let's C** has been entirely rewritten, and now uses the Mark Williams Lexicon format. This format has set the standard for language documentation on the Atari ST, and **Let's C** is the first C compiler to bring Lexicon format to the IBM PC.

How to use this manual

This manual is in nine sections. Section 1, which you are now reading, introduces **Let's C**.

Section 2 shows you how to install **Let's C** on your computer. It also introduces the **Let's C** shell, introduces the MicroEMACS screen editor, and shows you how to compile simple C programs.

Section 3 is entitled **C for Beginners**. If you are new to the C programming language, this section will introduce you to C. It is not a full tutorial on C, but it will show you the basics of C programming, so you will be better able to follow the rest of this manual and use the example programs in it.

Section 4 introduces compiling with **Let's C**. It describes the options to the compiler controller **cc**, and shows you how to compile using different memory models and different formats. Technical issues that involve the i8086 microprocessor and MS-DOS are also discussed.

Section 5 is a tutorial on the MicroEMACS screen editor. It introduces most of the MicroEMACS commands and includes exercises to help sharpen your skills at editing programs.

Section 6 is a tutorial on **make**, the Mark Williams programming discipline. **make** is one of the most useful tools available for constructing and maintaining large, intricate programs. This section describes **make**, from building relatively simple programs to using **make** to control work other than compiling C programs.

Section 7 presents some questions and answers about **Let's C**. New users of **Let's C** often ask the same questions about how to use it; so if you have a question, look here first. You could well find the answer you need.

Section 8 lists all of the error messages that the **Let's C** compiler, assembler, and utilities can produce. Many entries have hints to help you correct or avoid the error that the message describes.

Finally, section 9 is the Lexicon. This is by far the largest part of the manual. The Lexicon contains several hundred individual entries; each describes a command, a function, defines a C technical term, or gives you other useful information. All of the Lexicon's entries are in alphabetical order, and are designed to be easily used. For example, if you want information on how to use the STDIO routines, simply turn to the entry in the Lexicon on **STDIO**; there, you will find a list of all the STDIO routines, a description of each, and instructions on how to use them. Or, if you want information on how **Let's C** encodes floating point numbers, simply turn to the entry on **float**. There, you will find a full description of floating point numbers. Many Lexicon entries have full C programs as examples; all have cross-references to related entries.

The opening sections of this manual will refer constantly to the Lexicon. If you are unfamiliar with a technical term used in this manual, look it up in the Lexicon. Chances are, you will find a full explanation. If you are not sure how to use the Lexicon, look up the entry for **Lexicon** within the Lexicon. This will help you get started.

Finally, the back of the manual lists the Lexicon's entries sorted by category, and gives an index.

User registration and reaction report

Before you continue, fill out the User Registration Card that came with your copy of **Let's C**. When you return this card, you become eligible for direct telephone support from the Mark Williams Company technical staff, and you will automatically receive information about all new releases and updates.

If you have comments or reactions to the **Let's C** software or documentation, please fill out and mail the User Reaction Report included at the end of the manual. We especially wish to know if you found errors in this manual. Mark Williams Company needs your comments to continue to improve **Let's C**.

Technical support

Mark Williams Company provides free technical support to all registered users of **Let's C**. If you are experiencing difficulties with **Let's C**, outside the area of programming errors, feel free to contact the Mark Williams Technical Support Staff. You can telephone during business hours (Central time), or write. This support is available *only* if you have returned your User Registration Card for **Let's C**.

If you telephone Mark Williams Company, please have at hand your manual for **Let's C**. Please collect as much information as you can concerning your difficulty before you call. If you write, be sure to include the product serial number (from the sticker on the back of this manual) and your return address.

Bibliography

The following books may be helpful in developing your skills with C. This list also contains all books that are referenced in this manual. It is by no means exhaustive; however, it should prove helpful to both beginners and experienced programmers.

American National Standards Institute: *Draft Programming Language C (October 1986 Draft)*. Washington, D.C.: X3 Secretariat, Computer and Business Equipment Manufacturers Association, 1986.

AT&T Bell Laboratories: *The C Programmer's Handbook*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985.

Chirlin, P.M.: *Introduction to C*. Beaverton, Or.: Matrix Publishers, Inc., 1984.

Derman, B. (ed.): *Applied C*. New York: Van Nostrand Reinhold Co., Inc., 1986.

Feuer, A.R.: *The C Puzzle Book*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1982.

Gehani, G.: *Advanced C: Food for the Educated Palate*. Rockville, Md.: Computer Science Press, 1985.

Hancock, L.: Krieger, M.: *The C Primer*. New York: McGraw-Hill Book Publishers, Inc., 1982.

Harbison, S.; Steele, G.: *C: A Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Hogan, T.: *The C Programmer's Handbook*. Bowie, Md.: Brady Publishing, 1984.

Kelley, A.; Pohl, I.: *C by Dissection: The Essentials of C Programming*. Menlo Park, Ca.: The Benjamin/Cummings Publishing Company, Inc., 1987.

Kernighan, B.W.; Ritchie, D.M.: *The C Programming Language*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1978.

Kernighan, B.W.; Plauger, P.J.: *The Elements of Programming Style*, ed. 2. New York: McGraw-Hill Book Co., 1978.

Kochan, S.G.: *Programming in C*. Hasbrouck Heights, N.J.: Hayden Book Co., Inc., 1983.

Knuth, D.E.: *The Art of Computer Programming*, vol. 1: *Basic Algorithms*. Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 2: *Seminumerical Algorithms*. Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 3: *Sorting and Searching*. Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Plum, T.: *Learning to Program in C*. Cardiff, N.J.: Plum Hall, Inc., 1983.

Plum, T.: *C Programming Guidelines*. Cardiff, N.J.: Plum Hall, Inc., 1984.

Plum, T.; Brodie, J.: *Efficient C*. Cardiff, NJ: Plum Hall, Inc., 1985.

Purdum, J.: *C Programming Guide*. Indianapolis: Que Corp., 1983.

Purdum, J.; Leslie, T.C.; Stegemoller, A.L.: *C Programmer's Library*. Indianapolis: Que Corp., 1984.

Traister, R.J.: *Programming in C for the Microprocessor User*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

Traister, R.J.: *Going from BASIC to C*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

Vile, R.C., Jr.: *Programming in C with Let's C*. Glenview, IL: Scott, Foresman and Company, 1988.

Waite, M.; Prata, S.; Martin, D.: *C Primer Plus*. Indianapolis: Howard W. Sams, Inc., 1984.

Weber Systems, Inc.: *C Language User's Handbook*. New York: Ballantine Books, 1984.

Zahn, C.T.: *C Notes*. New York: Yourdan Press, 1979.

i8086/MS-DOS information

Duncan, R.: *Advanced MS-DOS: The Microsoft guide for assembly language and C programmers.* Redmond, WA: Microsoft Press, 1986. *Recommended.*

IBM Corporation: *Technical Reference, Personal Computer XT*. Boca Raton, FL: International Business Machines Corporation, 1983.

Intel Corporation: *8086 Relocatable Object Module Formats*. Document No. 121748-001. Santa Clara, CA: Intel Corporation.

Intel Corporation: *8086/8087/8088 Assembly Language Reference Manual for 8086-Based Development Systems*. Santa Clara, CA: Intel Corporation, 1983.

Intel Corporation: *iAPX 286 Programmer's Reference Manual*. Santa Clara, CA: Intel Corporation, 1985.

Microsoft Corporation: *MS-DOS Technical Reference Encyclopedia.* Bellevue, WA: Microsoft Press, 1986.

Norton, P.: *The Peter Norton Programmer's Guide to the IBM PC.* Bellevue, WA: Microsoft Press, 1985.

Young, M.: *Performance Programming Under MS-DOS.* Alameda, CA: SYBEX, Inc., 1987. *Recommended.*

Installing and Running Let's C

This section describes how to install **Let's C** onto your computer, and how to use it to compile simple programs.

Installing Let's C

Before you can use **Let's C**, you must *install* it on your computer. As **Let's C** comes to you, its files are fitted together to save space on its disks, not to run conveniently. This helps us lower our costs, to give you **Let's C** at the lowest possible price; however, it also means that before you can use the compiler you must recopy the files into organized groups.

To install **Let's C**, you must copy files from the distribution disks onto either a hard disk or, if you don't have a hard disk, a set of floppy disks. To make this job easy for you, **Let's C** includes the utility **install**, which does the copying for you. By running **install** and answering a few simple questions, you can build a working copy of **Let's C** on your system in a few minutes.

If you have a hard disk, use the directions in the section *Installing Let's C onto a hard disk*. If you do not have a hard disk, skip below to the section *Installing Let's C onto a floppy-disk system*, and follow the directions there.

Installing Let's C onto a hard disk

To begin, log into drive C on your system. You can do so by typing **c** at the MS-DOS prompt. On nearly all computers, drive C is the hard disk.

Now, insert **Let's C**'s distribution disk 1 into floppy drive A and type the following command:

```
a:install
```

In a moment, **install** will begin to work. It will print some information on your screen, and then ask you the following question:

```
Do you have a hard disk?
```

Type 'y', for "yes".

install will then ask you:

```
Do you wish to install all the files?
```

Answer 'y'.

install will now ask you in which directories you wish to install the files, as follows:

```
Where do you want the executable programs?
Where do you want the header files?
Where do you want the libraries?
Where do you want the sample programs?
Where do you want the source files?
Where do you want the temporary files?
```

After each question, type **<return>**, which signifies the default setting. Later, you may wish to re-install **Let's C** into other directories, but you should use the default settings until you gain some familiarity with **Let's C**.

install will now copy the files from the distribution disk onto your hard disk. Depending on how fast your system is and how fast your hard disk is, this can take from five to 15 minutes. When all the files from one disk are copied, **install** will ask for the next disk, until all files are copied.

When **install** exits, it will print some information on the screen, and then an instruction of the form:

```
set CCHEAD=@c:\lib\ccargs
```

Copy down this instruction. You should then write this instruction into the file **autoexec.bat** on your MS-DOS boot disk. This instruction tells **Let's C** where you have stored all of its components, so it can find everything correctly. This will be discussed below, in the section entitled *Setting your computer's environment*.

Once all the files are copied, place your copy of the MS-DOS disk into the floppy disk drive. If you have version 3.2 of MS-DOS, you have two MS-DOS disks; insert the disk labelled "supplemental programs". Now, type the command:

```
copy a:link.exe \bin
```

This copies the MS-DOS linker MS-LINK into directory **\bin** on your hard disk. **Let's C** uses MS-LINK to link its executable files, so MS-LINK must be copied into a directory where **Let's C** can find it. If you did not use the default directory names, copy **link.exe** into the directory where you stored **Let's C**'s executable files.

That's all there is to it. **Let's C** is now installed on your hard disk. Now, skip below to the section entitled *Setting your computer's environment*. This will tell you how to set the environment, so **Let's C** can work efficiently.

Installing Let's C onto a floppy-disk system

Let's C is too large to fit onto a single floppy disk. Therefore, if your system does not have a hard disk you must install **Let's C** onto a set of seven floppy disks, as follows:

Disk 1 The **shell** disk. This disk holds MWS, the Mark Williams shell. You will use it only to boot MWS, then put it away.

Disk 2 The compiler disk for standard-sized programs. You should use this disk to compile programs that are of normal size and complexity. This disk holds the compiler controller **cc**, the phases of compilation, the MicroEMACS screen editor, and other tools used during compilation.

Disk 3 The compiler disk for unusually large programs. If you have written a program thatis unusually large or complex, and it will not compile correctly with disk 2, use disk 3 instead. This disk has all of the files that appear on disk 2, plus LARGE-model versions of the compiler phases **cc0** and **cc2**.

Disk 4 This disk holds the SMALL-model libraries. You will need to copy MS-LINK onto this disk.

Disk 5 This disk holds the LARGE-model libraries. You will also need to copy MS-LINK onto this disk.

Disk 6 This disk holds the **Let's C** commands and utilities. For a fuller description of the commands, see the Lexicon entry for **commands**.

Disk 7 This disk holds the sample programs and source code that comes with **Let's C**. This includes the full source code for the MicroEMACS screen editor.

To begin, format eight new floppy disks. Label them respectively as follows:

```
1. shell
2. normal compiler
3. compiler for large programs
4. SMALL-model libraries
5. LARGE-model libraries
6. commands
7. source code and samples
```

Now, at the MS-DOS prompt, type **B:**. This will log into drive B on your machine. Insert **Let's C**'s distribution disk 1 into drive A; type the following command:

```
a:install
```

In a moment, **install** will begin to execute, and will print some information on your screen. It will then ask you this question:

```
Do you have a hard disk?
```

Answer 'n', for "no".

install will then ask:

```
Do you wish to install all of the files?
```

Type 'y', for "yes".

install will now ask you in which directories you wish to install the files, as follows:

```
Where do you want the executable programs?
Where do you want the header files?
Where do you want the libraries?
Where do you want the sample programs?
Where do you want the source files?
Where do you want the temporary files?
```

After each question, type **<return>**, which accepts the default setting. Later, you may wish to re-install **Let's C** into other directories, but you should use the default settings until you gain some familiarity with **Let's C**.

install will now tell you:

```
Insert the shell disk into drive B.
```

Insert the formatted floppy disk that you labelled "shell" into drive B. **install** will copy the appropriate files onto it. When **install** needs a new source disk, it will prompt you for it.

install will go through this procedure for each of the seven floppy disks that you will be building. It will prompt you when to change disks, and tell you which disk to insert into drive A or drive B.

When **install** exits, it will print some information on the screen, and then an instruction of the form:

```
set CCHEAD=@a:\ccargs
```

Copy down this instruction. You should then edit this instruction into the file **autoexec.bat** on your MS-DOS boot disk. This instruction tells **Let's C** where you have stored all of its components, so it can find everything correctly. This will be discussed below, in the section entitled *Setting your computer's environment*.

When **install** has finished, you must do the following for each of the four library disks, disks 3 through 6. First, place your computer's MS-DOS disk into drive A. If you have MS-DOS version 3.2 or later, your copy of MS-DOS comes on two disks; insert the disk labelled "supplemental programs" into drive A. Then, place disk 4 (which holds the SMALL-model libraries) into drive B. Type the following command:

```
copy a:link.exe b:\bin
```

This command will copy the MS-DOS linker MS-LINK into directory **\bin** on your library disk. **Let's C** uses MS-LINK to link the executable files it creates, so MS-LINK must be copied into a directory where **Let's C** can find it.

Repeat this procedure for disk 5.

That is all there is to it: **Let's C** is now installed on your computer.

When you are finished, you may wish to recopy your installed disks, to save yourself the trouble of having to reinstall should something happen to your working copy.

Now, read the following section *Setting your computer's environment*, which tells you how to set your computer's environment so you can use **Let's C**.

Re-installing a portion of Let's C

If you wish, you can re-install just a portion of the compiler. When **install** asks

```
Do you wish to install all of the files?
```

answer 'n'. **install** will then prompt you for which portion, or portions, of **Let's C** you wish to install, and will then install it for you.

Setting your computer's environment

As you have probably noticed by now, **Let's C** is not just one program: it is a collection of more than 100 files and programs, which together form one of the most sophisticated C programming systems available at any price. Despite its complexity, **Let's C** is designed to work smoothly, and transform your source code into an executable file in the shortest possible time.

For **Let's C** to work at peak efficiency, however, you should pay some attention to your computer's *environment*. The environment is a set of variables that are available to all of the programs run on your computer. By setting the environment properly, you will help **Let's C** find all of its programs quickly to speed your work.

Setting the PATH

When you installed **Let's C**, all of its commands were copied into a directory called **\bin**. When you type a command into MS-DOS, MS-DOS looks for it in the directories named in the environmental variable **PATH**. To ensure that MS-DOS can find **Let's C**'s commands you must set the **PATH** so that it names the directory **\bin**.

To set the **PATH**, use the MS-DOS command **path**. For example, if you want keep your executable files in the directories **bin** and **mwc**, type:

```
PATH=C:\BIN;C:\MWC
```

If your computer does not have a hard disk, use the following form of the **path** command:

```
PATH=A:\BIN;A:\MWC;B:\BIN;B:\MWC
```

This tells MS-DOS to look for **cc** in the directories **bin** and **mwc** on both of your floppy disk drives. With the **PATH** set to this configuration, it does not matter which disk drive holds your compiler disk when you work: MS-DOS will find **cc** in either drive automatically.

When the **PATH** is set properly, you should copy the **path** command you used into the file **autoexec.bat**, on your computer's boot disk. Then, the **PATH** will be set automatically whenever you boot your computer.

Finding the ccargs file

When you installed **Let's C** onto your computer, the program **install** created a file called **ccargs**. This file contains the names of all of the directories in which **install** stored the elements of **Let's C**.

When you compile a program, **Let's C** reads **ccargs** and uses its entries to find all of the files it needs. If you have a floppy-disk system, **install** wrote **ccargs** into directory **a:** on disk 2 (the compiler disk). If you have a hard disk, **install** wrote **ccargs** into the directory in which you installed the **Let's C** libraries (the default is **c:\lib**).

You must set an additional environmental variable so that **Let's C** can locate **ccargs** when you compile a program: the environmental variable **CCHEAD**. As noted above, when **install** finished installing **Let's C** on your system, it printed on your screen an instruction of the form

```
set CCHEAD=@directory\ccargs
```

where *directory* is the name of the directory into which it wrote **ccargs**. For example, if you have a floppy-disk system, **install** printed the message

```
set CCHEAD=@a:\ccargs
```

whereas if you have a hard disk, it printed the message

```
set CCHEAD=@c:\lib\ccargs
```

Be sure that you copy this instruction into the file **autoexec.bat** on your MS-DOS boot disk. Once you have done this, reboot your system; this ensures that **CCHEAD** is set for your system.

That's all there is to it. When you copy the **set** commands for **PATH** and **CCHEAD** into **autoexec.bat** and reboot your system, you have set **Let's C**'s environment. **Let's C** is now ready to start working for you.

Editing ccargs

ccargs correctly describes where everything is stored on your computer. The default **ccargs** for a hard disk reads as follows:

```
-xcc:\bin\
-xlc:\lib\
-xtc:\tmp\
-Ic:\include\
```

The default **ccargs** for a floppy disk system reads as follows:

```
-xca:\bin\
-xla:\lib\
-Ia:\include\
-Z
```

If you later decide to move part of **Let's C** from one directory to another, you must edit **ccargs** to reflect this change, or **Let's C** will not know where you moved **Let's C**. You should edit **ccargs** as follows:

Executable files

If you move the executable files from where you installed them, change the line in **ccargs** that begins **-xc**. For example, if you have a floppy disk system and you move the executable files from directory **\bin** to directory **\mwc**, change the line

```
-xca:\bin\
```

to read

```
-xca:\mwc\
```

Libraries

If you move the libraries from where you installed them, you should change the line in **ccargs** that begins **-xl**. For example, if you have a hard disk and you move the libraries from directory **\lib** to directory **\library**, change the line

```
-xlc:\lib\
```

to read

```
-xlc:\library\
```

Header files

If you move the header files from where you installed them, you should change the line in **ccargs** that begins **-I**. For example, if you have a floppy disk system and you move the header files from directory **\include** to directory **\header**, change the line

```
-Ia:\include\
```

to read

```
-Ia:\header\
```

Temporary files

Finally, **ccargs** tells **Let's C** where to write your temporary files. This is set only in the version of **ccargs** that is used with a hard disk. If you decide to change where **Let's C** writes its temporary files, you must edit the line in **ccargs** that begins **-xt**. For example, if you want to write temporary files into directory **nowhere**, change the line

```
-xtc:\tmp\
```

to read

```
-xtc:\nowhere\
```

If you need more information on where **Let's C** looks for files, or how **Let's C** works in general, look at the Lexicon entry for **cc**. This will describe in some detail how **Let's C** works, and also describe other ways that you can change how **Let's C** looks for its files.

Using MWS, the Let's C command interface

Let's C includes an interface program, MWS, which is designed to help you develop programs more quickly and efficiently. MWS, which stands for "Mark Williams shell", accelerates the speed with which your programs work, and it has a display interface that helps you build commands with ease.

To invoke MWS, simply do the following. If you do not have a hard disk, insert disk 1 (the "shell" disk) into disk drive A. Then, at the MS-DOS prompt type:

```
MWS
```

In a moment, the screen will clear and the MWS main menu will appear:

```
Let's C Version 4.0
(c) 1987 Mark Williams Company, Chicago

+----------------+
|  Edit          |
|  Compile       |
|  Run           |
|  Debug         |
|  Make          |
|  Buffers       |
|  Quick DOS     |
|  !DOS Escape   |
|  New directory |
+----------------+









<return> select      <F1> more help
<→> use arrow keys   <esc> exit menu
```

As you can see, the entry for **Edit** on the menu is marked by a reverse-video band, called the *cursor bar*. In this tutorial, the cursor bar will be shown as shading. The cursor bar indicates the entry in the main menu you wish to select. Try pressing the down-arrow key (↓) on the keypad. The cursor bar now covers the entry **Compile**. Each selection will be discussed in detail in the following sub-sections.

Note that if you do not want to bother with moving the cursor bar, you can pick an option simply by typing its first letter. For example, you can begin to edit a file simply by typing 'e'.

At the bottom of the screen are listed the commands that you can give MWS. As noted above, pressing the arrow keys moves the cursor bar up and down the menu. Pressing **<return>** tells MWS to select the menu entry that the cursor bar is highlighting. Pressing **<esc>** exits. If you are in a sub-menu, pressing **<esc>** returns you to the previous menu; whereas if you are already in the main MWS menu, pressing **<esc>** exits you from MWS altogether, and returns you to MS-DOS.

Finally, pressing the function key **<F1>** prints a help message. Each screen has its own help message, to guide you through MWS even if you do not have this manual available.

Editing a file

MWS includes a full-featured screen editor, called MicroEMACS. An *editor* is a program that lets you type text into your computer, store it on disk, then recall it from disk and change it. You will use an editor to type all of the programs that you compile with **Let's C**.

MicroEMACS allows you to divide the screen into sections, called *windows*, and display and edit a different file in each one. It has a full search-and-replace function, allows you to define keyboard macros, and has a large set of commands for killing and moving text. Also, MicroEMACS has a full help function for C programming. Should you need information about any macro or library function that is included with **Let's C**, all you need to do is move the text cursor over that word and press a special combination of keys; MicroEMACS will then open a window and display information about that macro or function.

Let's C includes both a compiled, binary version of MicroEMACS that is ready to use, and the full source code. We invite you to examine the code, modify it, and enhance MicroEMACS to suit your preferences.

For a list of the MicroEMACS commands, see the Lexicon entry for **me**, the MicroEMACS command. A following section of this introduction gives a full tutorial on MicroEMACS. In the meantime, however, you can begin to use MicroEMACS by learning a half-dozen or so commands.

MWS lets you access the MicroEMACS screen directly through its display interface. To edit a file through MWS, do the following. First, if you do not have a hard disk, remove the "shell" floppy disk from drive A and put it aside. Place disk 2, the "compiler" disk, into drive A. Then, press the up-arrow key (↑) until the cursor bar covers the entry **Edit**. Press **<return>**. This *selects* the edit feature; that is, it tells MWS that you wish to use the MicroEMACS editor to edit a file.

In a moment, MWS redraws the screen as follows:

```
Edit

      +======================================+
      | me                                   |
      +======================================+

      +============+
      |  Execute   |
      |  Files     |
      |  New File  |
      +============+









      <return> select      <F1> more help
      <→> use arrow keys   <esc> exit menu
```

The cursor bar is now positioned over the selection **Execute**. Press the ↓ key two times, so the cursor bar is positioned over **New File**. Press **<return>**. The menu disappears, and MWS prompts you to enter the name of the file you wish to create.

Type **hello.c**. Note that the name **hello.c** appears in the long box at the top of the screen; this box is called the *command box*, because it is where MWS builds your command. Note, too, that the cursor bar has returned to the entry **Execute** on the menu.

Now press **<return>**. This tells MWS to execute the command that is displayed in the command box. The screen again clears: you have just invoked the MicroEMACS editor to edit the file **hello.c**.

Now, type the following text, as it is shown here. If you make a mistake, simply backspace over it and type it correctly; the backspace key will wrap around lines:

```
main()
{
        printf("hello, world\n");
}
```

When you have finished, *save* the file by typing **<ctrl-X><ctrl-S>** (that is, hold down the control key and type 'X', then hold down the control key and type 'S'). MicroEMACS will tell you how many lines of text it just saved. Exit from the editor by typing **<ctrl-X><ctrl-C>**.

Let's C

When you have exited from MicroEMACS, MWS redisplays the edit menu, with the cursor bar positioned over **Execute**. Note that MWS remembers the last command you built for each of the commands on the main menu (that is, the last **Edit** command, the last **Compile** command, and so on). If you do not wish to build a new command, you can re-execute the last command you built by simply pressing **<return>**.

Now, type **<return>**. MWS will invoke MicroEMACS with the last file you edited, **hello.c**. The text of the file you just typed is now displayed on the screen. Try changing the word **hello** to **Hello**, as follows: First, type **<ctrl-N>** That moves you to the *next* line. (The command **<ctrl-P>** would move you to the *previous* line, if there were one.) Now, type the command **<ctrl-F>**. As you can see, the cursor moved *forward* one space. Continue to type **<ctrl-F>** until the cursor is located over the letter 'h' in **hello**. If you overshoot the character, move the cursor *backwards* by typing **<ctrl-B>**.

If you prefer, you can also move the cursor by pressing the arrow keys.

When the cursor is correctly positioned, delete the 'h' by typing the *delete* command **<ctrl-D>**; then type a capital 'H' to take its place.

With these few commands, you can load files into memory, edit them, create new files, save them to disk, and exit. This just gives you a sample of what MicroEMACS can do, but it is enough so that you can begin to do real work.

Now, again *save* the file by typing **<ctrl-X><ctrl-S>**, and exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**. Again, the screen shows the **Edit** screen. Type **<esc>**; as shown at the bottom of the screen, this will exit you from the **Edit** menu, and so return you to the main menu.

Just as a reminder, the following table gives the MicroEMACS commands presented above:

<ctrl-N> or ↓	Move cursor to the *next* line
<ctrl-P> or ↑	Move cursor to the *previous* line
<ctrl-F> or →	Move cursor *forward* one character
<ctrl-B> or ←	Move cursor *backward* one character
<ctrl-D>	*Delete* a character
<ctrl-X><ctrl-S>	*Save* the edited file
<ctrl-X><ctrl-C>	Exit from MicroEMACS

Simple compiling

Now that you have prepared a C program with the MicroEMACS editor, the next step is to compile it into executable form.

To compile a program under **Let's C**, you must use the command **cc**. MWS's display interface lets you easily construct commands for **cc** to execute. To see how this works, press the ↓ key once, so the cursor bar is positioned over **Compile**. Press **<return>**, to select this option.

MWS redraws the screen so it appears as follows:

```
Compile
      +----------------------------------------+
      | cc                                     |
      +----------------------------------------+

      +------------+
      |  Execute   |
      |  Options   |
      |  Files     |
      +------------+









      <return> select      <F1> more help
      <→> use arrow keys   <esc> exit menu
```

The cursor bar is positioned over **Execute**. Press the ↓ key, to move the cursor bar to **Options**; then press **<return>**. The screen will be redrawn to appear as follows:

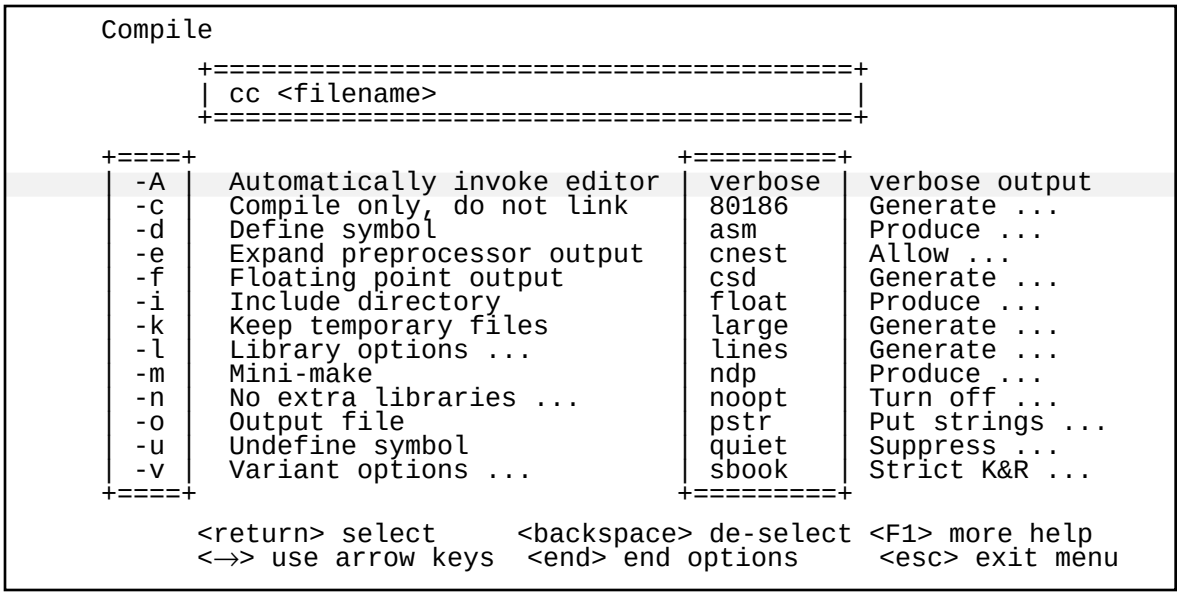
```
Compile
      +========================================+
      | cc <filename>                          |
      +========================================+

+====+
| -A |  Automatically invoke editor
| -c |  Compile only, do not link
| -d |  Define symbol
| -e |  Expand preprocessor output
| -f |  Floating point output
| -i |  Include directory
| -k |  Keep temporary files
| -l |  Library options ...
| -m |  Mini-make
| -n |  No extra libraries ...
| -o |  Output file
| -u |  Undefine symbol
| -v |  Variant options ...
+====+

      <return> select     <backspace> de-select <F1> more help
      <→> use arrow keys <end> end options     <esc> exit menu
```

cc's options cannot all fit onto one screen; if you press the ↓ key until it is at the bottom of the menu, you can then scroll through the options that are not initially shown on the screen. The options with ellipses "…" after their descriptions have a menu of sub-options associated with them.

For the present exercise, type the ↓ key until the cursor bar is positioned over the entry **-v**, for *variant* options. Press **<return>**. A second set of options appears on the screen, which now appears as follows:



The cursor bar is over the **verbose** option. Type **<return>**; this selects the **verbose** option, which tells you what steps the compiler is executing as it compiles. The command box now reads:

```
cc -V <filename>
```

Type **<end>**. The variant options box disappears, and the cursor is repositioned over the **-v** option. Type **<end>** again. The options box has disappeared, and you are back in the **Compile** menu.

Now, press the ↓ key, to position the cursor bar over **Files**. Press **<return>**. In a moment, two boxes will appear. One displays all of the files that have the suffix **.c**, and the other displays all of the files that have the suffix **.obj**, if any. To move between the boxes, press **<tab>**, as shown at the bottom of the screen. Press the ↓ until the cursor bar is positioned over **hello.c**. Press **<return>**. This completes the construction of the **cc** command line, as shown in the command box.

Note that you must select a file by moving the cursor bar to its name and pressing return; you cannot select a file by typing the first character in its name, because more than one file could use that character.

Press **<end>** to tell MWS that you have selected all of the files you want. MWS then returns you to the compile menu, with the cursor bar positioned over **Execute**. Press **<return>**, to begin execution of the command you have built.

Compilation now begins automatically. The switch **-V** tells **cc** to describe each step in compilation.

If your computer does not have a hard disk, **Let's C** will print the following prompt on your screen when it comes time to link your program:

```
Insert floppy disk 3 (SMALL-model, i8087-sensing libraries)
```

Open drive A, remove the compiler disk, and insert floppy disk 3. If your computer does have a hard disk, you will not need to do this. As noted earlier, when you installed **Let's C**, this disk holds the libraries for SMALL-model programs that sense the presence of the i8087 co-processor. For more information on how these libraries work, see the Lexicon entries for **library**, **model**, and **i8087**. When you have inserted this disk, press **<return>**. **Let's C** will now invoke MS-LINK and link your program.

As you can see, **cc** guides the program through all phases of compilation, invokes the linker, and links in all necessary routines from the libraries to produce a file called **hello.exe** that is ready to execute.

When compilation is finished, you will be returned to the MWS main menu.

Running a program

The next step is to run the program you just compiled. MWS lets you run any program you have compiled with **Let's C**, or in fact any executable program whatsoever, through its display interface.

If your computer does not have a hard disk, before you begin this step you should open drive A, remove floppy disk 3 (a libraries disk), and reinsert floppy disk 2 (the compiler disk). If your computer does have a hard disk, you do not need to do this.

To run **hello.exe**, which you created when you compiled **hello.c**, press the ↓ key until the cursor bar is positioned over **Run**. Then, press **<return>** to select this option. The main menu disappears, and MWS redraws the screen as follows:

```
Run
      +========================================+
      | <filename>                             |
      +========================================+

      +============+
      |  Execute   |
      |  Arguments |
      |  Files     |
      +============+









      <return> select      <F1> more help
      <→> use arrow keys   <esc> exit menu
```

As before, you can use the arrow keys to move the cursor bar up and down the menu. If you press **<return>** with the cursor positioned over **Execute**, MWS will re-execute the last command you built, if any. If your program needs arguments, select the **Arguments** option, which will prompt you to enter them. If your command does not take any arguments, simply type **<return>**.

Press the ↓ key once. The cursor bar is now positioned over **Files**. Press **<return>**. MWS draws a box that holds all executable files in the current directory—that is, all files that have the suffix **.exe**. Press the ↓ key until the cursor bar covers **hello.exe**. Press **<return>**. As you can see, the command box now shows **hello.exe**. Press **<esc>** to exit from this menu.

You have returned to the **Run** menu, with the cursor bar over **Execute**. Press **<return>**. MWS now executes the command shown in the command box, **hello.exe**. It executes, and prints on your screen **hello, world**. To return to MWS's main menu, press any key.

As you can see, it is easy to create, compile, and execute a program through the MWS menu interface.

Let's C

Quick DOS and !DOS options

MWS gives you two ways to give commands directly to MS-DOS while still enjoying MWS's acceleration of your programs.

To give just one command to MS-DOS, press the ↓ key until the cursor bar is positioned over **Quick DOS**. Press **<return>**. At the bottom of your screen appears the prompt

```
DOS command:
```

Type **dir**. MS-DOS executes **dir** and prints the contents of the current directory on your screen. When **dir** has finished executing, press any key to return to MWS's main menu.

If you want to give a number of commands directly to MS-DOS, press the ↓ key until the cursor bar is positioned over **!DOS Escape**. Press **<return>**. The main menu again disappears, and your MS-DOS prompt appears on the screen. MS-DOS is now ready to receive any number of your commands. If you are familiar with the **cc** command or MicroEMACS, you can now type these commands directly to MS-DOS. MWS does not require you to use its display interface, but it will still accelerate your work.

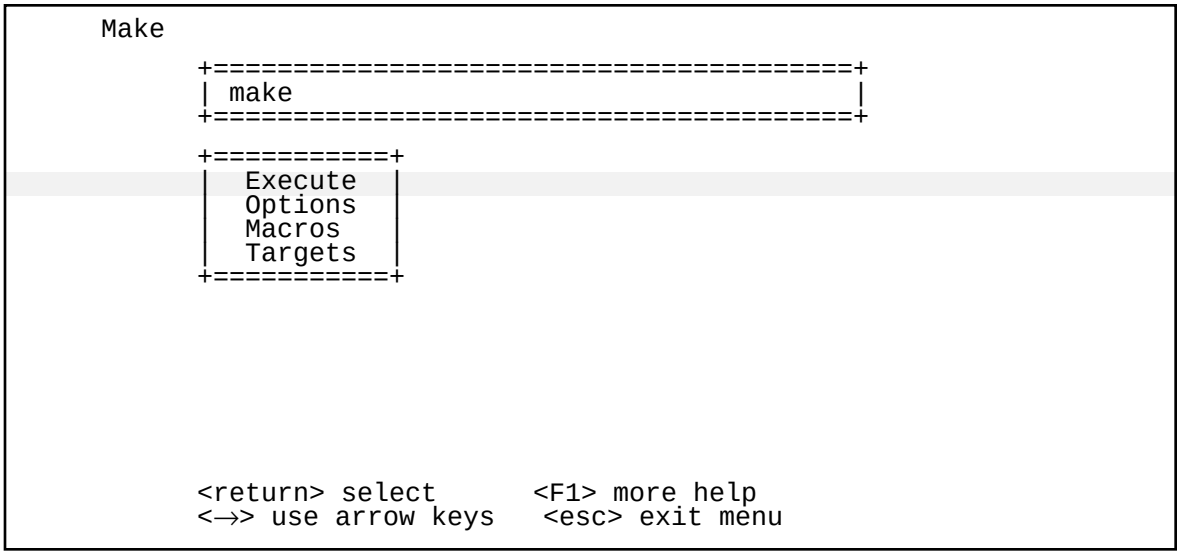
When you are done working with MS-DOS, type **exit**. The MS-DOS prompt will disappear, and the MWS main menu will reappear.

Using the make programming discipline

MWS gives you quick access to **make**, the Mark Williams programming discipline.

make helps you build large, complex programs. It reads a **makefile** that you prepare, and follows the **makefile**'s descriptions to build your program. If you need more information on **make**, see the entry for **make** in the Lexicon, and see the tutorial on **make** that appears in section 5 of this manual. The tutorial also describes in detail how to write a **makefile**.

To invoke **make** through MWS, press the ↓ key until the cursor bar covers **Make**. Type **<return>**. The main menu disappears, and the screen appears as follows:



The cursor bar is over the top entry in the menu, **Execute**. MWS remembers the last command you gave **make**, if any; so, if you press **<return>** MWS will automatically execute the last **make**

command you issued.

If you wish to construct a new **make** command, the first step is to change **make**'s default macros so that **make** will produce the sort of executable program that you want. Note that this step often is not necessary. For more information on **make**'s macros and the files in which they are kept, see the Lexicon entry for **make**; also see the **make** tutorial.

To redefine a macro, press the ↓ key twice, to move the cursor bar to **Macros**; then press **<return>**. MWS will prompt you to enter the new macro; what you type will then be used instead of any macro that has the same name.

Likewise, should you wish to change the default targets that **make** constructs, press the ↓ key until the cursor bar covers **Targets**; then press **<return>**. MWS will prompt you to type the new target or targets that you want to build. For more information on what a target is and how **make** builds one, see the tutorial for **make**, or see the Lexicon entry for **make**.

The next step is to construct the **make** command line. Press the ↑ key to move the cursor bar to **Options**. Press **<return>**. The screen is redrawn to offer you the options for **make**, as follows:

```
Make

      +=======================================+
      | make                                  |
      +=======================================+

+====+
| -d |  (Debug) Verbose output
| -f |  Alternate make filename
| -i |  Ignore error returns and continue
| -n |  Show commands but do not execute
| -p |  Print macro and target descriptions
| -q |  Query target status
| -r |  Do not use built-in rules
| -s |  Silent running
| -t |  Touch all targets to current time
+====+



      <return> select     <backspace> de-select <F1> more help
      <→> use arrow keys  <end> end options     <esc> exit menu
```

To select an option, press the ↓ key until the cursor bar is positioned over it, and then press **<return>**. The option will be written into the command box at the top of the screen. Should you select an option by accident, press **<backspace>**; the option will be erased from the command box at the top of the screen. If you select the **-f** option, MWS will prompt you for the name of the file that you wish to use in place of **makefile**. For more information on **make**'s options, see the entry for **make** in the Lexicon, or see the tutorial on **make** later in this manual.

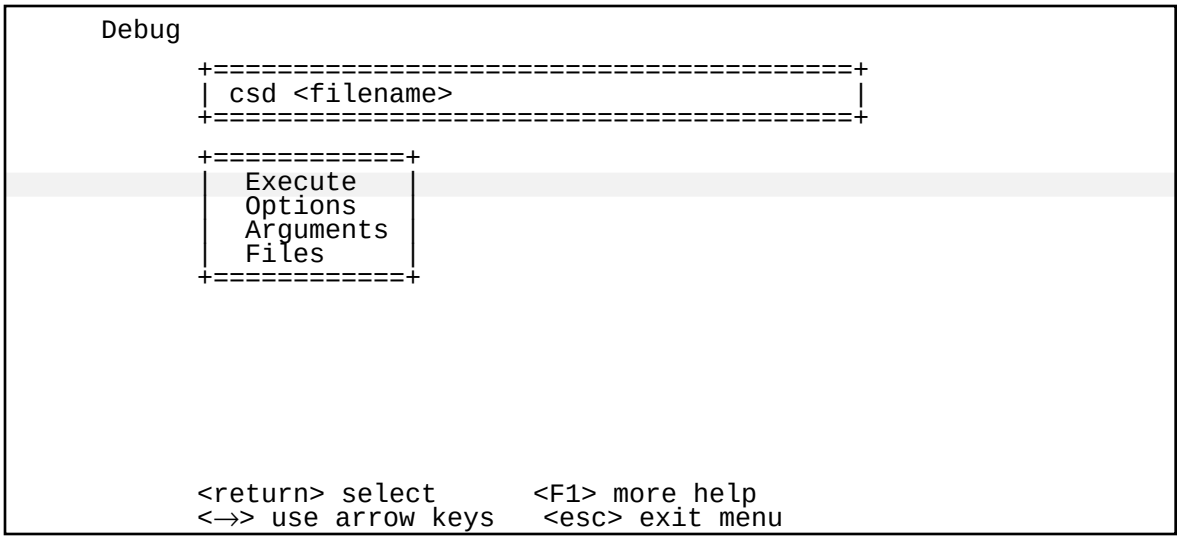
When you have selected all of the options that you want, press **<end>**. You will be returned to the main **Make** menu, and the cursor bar will be positioned over **Execute**. Press **<return>**; MWS will then execute the **make** command that you have built, which appears in the command box at the top of the screen. **make** will look for **makefile**, read it, and begin to build your program. When **make** is finished, you will be returned to the MWS main menu.

Using csd, the C source debugger

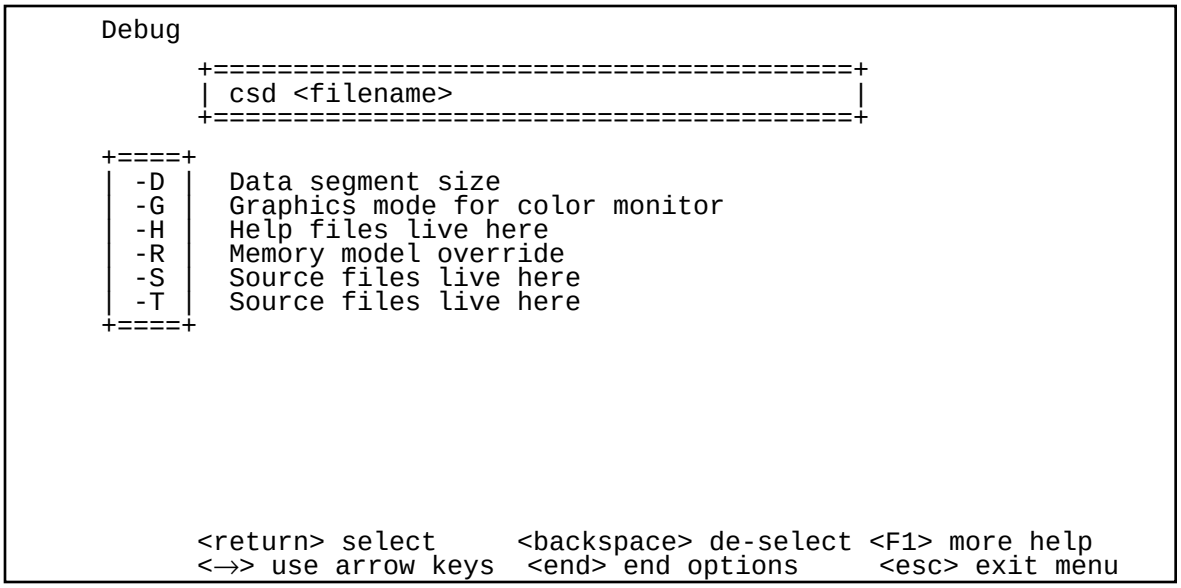
MWS also helps you to invoke **csd**, the Mark Williams C source debugger. **csd** is an invaluable tool to a programmer, for it allows you to debug programs while using your C source code, instead of having to use listings in assembly language. Note that **csd** is not included with **Let's C**, but it is available as a separate product for use with **Let's C**.

If your computer does not have a hard disk, you must insert the disk that came with your copy of **csd** into drive A before you can begin to work with **csd**.

To invoke **cs d**,press the ↓ key until the cursor bar is positioned over **Debug**. Press **<return>**. MWS redraws its screen as follows:



If you wish to construct a new **csd** command, press the ↓ key once, to position the cursor bar over **Options**; then type **<return>**. MWS will redraw its screen to display the available options, as follows:



If you select either **-H** or **-S**, MWS will prompt you to enter the name of the directory where you have stored these files. For more information on **csd**'s options, see your **csd** manual.

As before, if you select an option by accident, press **<backspace>**. This will erase the option from the command box at the top of your screen. When you have selected all of the options that you want, press **<end>**. The list of options will disappear from the screen, and the original **Debug** menu reappear.

To complete your **csd** command, press the ↓ key twice; this moves the cursor bar to **Files**. Press **<return>**. MWS will then draw a box that contains all of the executable files in the current directory. You can select one by moving the cursor bar to it, and then pressing **<return>**.

Remember that to debug a file with **csd**, it must first have been compiled with the **-VCSD** option to the **cc** command line. This writes special debugging information into the executable file.

When you have selected the program you wish to debug and have pressed **<return>**, MWS will write the file name into the command box, and return you to the **Debug** menu; the cursor bar is positioned over **Execute**. Press **<return>** to invoke **csd** and execute the command you have just built. You can then debug your program with **csd** just as it is described in the **csd** manual.

When you have finished your debugging session and have exited from **csd**, you will return to the MWS main menu. MWS will then be ready help you build another command.

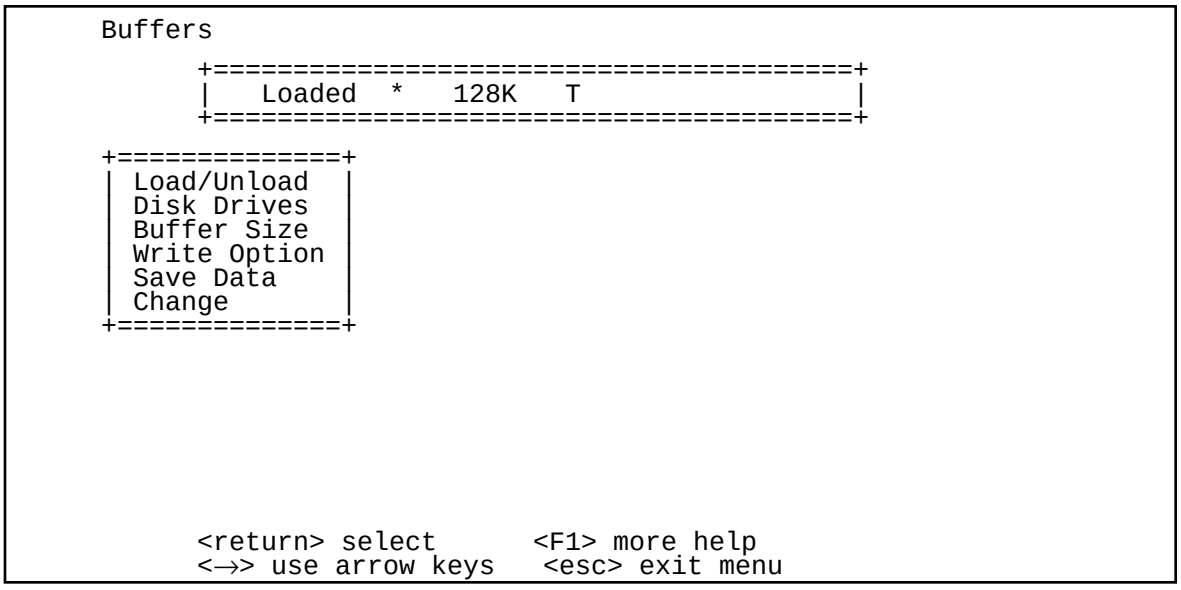
Resetting the buffers

As noted earlier, MWS not only gives you an easy way to build commands for **Let's C**: it also contains an accelerator that speeds up your software. The accelerator uses a buffering system that reduces the number of times that programs need to read the disk drive. Because reading the disk drive is the slowest part of any program, reducing the number of times the disk must be read increases the speed with which the program operates.

At times, you may need to change how MWS performs its buffering. To do so, use the **Buffers** command on MWS's main menu.

Let's C

To begin, press the ↓ key until the cursor bar is positioned at the entry **Buffers**. Press **<return>**. MWS will draw the **Buffers** screen, as follows:



Each command is described below.

Load/Unload

This tells MWS to load or unload the accelerator. Note that this command is a *toggle*: if the accelerator is unloaded, then pressing **<return>** tells MWS to load it, and vice-versa. Try pressing **<return>**. You will see that the entry in the command box will flip from **Loaded** to **Unloaded**.

Note, too, that this command does *not* take effect immediately. To unload or load the accelerator, you must exit from MWS and then re-enter it.

Disk Drives

This command tells MWS which disk drives you want accelerated. The default is '*', which indicates that all drives are accelerated.

To change the settings, press the ↓ key until the cursor bar is positioned over **Disk Drives**. Press **<return>**. MWS will prompt you for the name of a disk drive. Type **A**, to indicate that you want to accelerate disk drive A. Note that the asterisk in the command box has been replaced with **A:**. The prompt remains on the screen: if you wish, you can name any number of disk drives. To end this session, press **<return>** without typing anything. MWS will now accelerate only disk drive **A:**.

Now, re-invoke this command by moving the cursor bar to **Disk Drives** and pressing **<return>**. Type '*', and then press **<return>** twice. MWS has now resumed accelerating all disk drives.

Buffer Size

The accelerator reserves a portion of memory to buffer what it reads from your disk drives. By default, the size of its buffer is 128 kilobytes, as shown in the command box. If your system has limited amounts of memory, you may wish to decrease this amount. On the other hand, if your system has memory to spare you may wish to increase the size of the buffer: the larger the buffer is, the more your software will be accelerated.

To change the size of the buffer, press the ↓ key until the cursor bar is positioned over **Buffer Size**, and then press **<return>**. MWS will prompt you for the new buffer size. If you change your mind and decide to leave the buffer unchanged, simply press **<return>** without entering anything.

Note that this command, like the **Load/Unload**, does *not* take effect immediately. You must leave MWS and then re-enter it before you can begin working with a different sized buffer.

Write Option

The MWS accelerator not only speeds up the rate with which your programs read the disk: it also speeds up the rate at which they *write* data to the disk.

The accelerator offers you three ways to save your data to the disk: **Memory**, **Timed Save**, and **Disk**. The **Memory** option stores all data in memory. No data are written to the disk until you choose to save them with the **Save Data** command, which will be described in a moment. The **Disk** option writes all data directly to disk, and does not buffer data at all. Note that the **Memory** option is faster than the **Disk**, but not as safe, because an accident could cause you to lose the data stored in the buffer.

The **Timed Save** option combines features of the **Memory** and **Disk** options. Data are stored in memory, but they are automatically written to disk every five minutes. Thus, you have the speed of in-memory storage, plus the safety of saving data to disk. This is the default option.

To change the write option, press the ↓ key until the cursor bar is over **Write Option**, and then press **<return>**. A menu will appear that displays the three write options. Move the cursor bar to the one you want and then press **<return>**. The option you select will be shown in the command box, and you will be returned to the **Buffers** menu.

Save Data

This command writes to disk all data that MWS has stored in its buffer. To use it, simply press the ↓ key until the cursor bar is positioned over **Save Data**, and then press **<return>**. This command has no options: it will simply write the data, and return you to the **Buffers** menu.

You should use this command to save your data before you turn off your machine or before you leave MWS, should you be using the **Memory** option.

Change Disk

The last command, **Change Disk**, *must* be used before you change the floppy disk in a disk drive that is being accelerated. This command writes to disk all of the data that MWS has stored in its buffer, and then empties the buffer to make it ready for the new disk.

If you do not use this command before you change a floppy disk, you could lose data. This bears repeating: *You must use this command before you change a floppy disk in a drive that is being accelerated, or you could lose data.*

To use this command, simply press the ↓ key until the cursor bar is positioned over **Change Disk**. Press **<return>**. Any saved data will be written out, the buffer will be emptied, and MWS will be ready to accept a new floppy disk.

To return to the main MWS menu, press **<esc>**.

As you can see, MWS's accelerator is easy to use. With a few simple commands, you can alter its settings to suit your preferences. MWS will remember these settings, and use them automatically in the future.

Let's C

Where to go from here

The following bibliography lists reference books on C and on the i8086 processor. This list includes both primers and references for advanced programmers. This list is not exhaustive, but you will find it helpful should you need detailed information on a topic.

Section 2, *C for Beginners*, introduces the C language to users who are new to C. If you are experienced with C you may wish to skip this section, but if you are a novice at C programming you may find it helpful.

After *C for Beginners* is a section on *Advanced Compiling*. This discusses many of the compiler's options that were just mentioned here. It also discusses some technical issues on the i8086 microprocessor that both experienced programmers and novices will find helpful.

Finally, if you need more information on any command, library function, C keyword, or technical term, check the Lexicon.

C for Beginners

In the last few years, C has grown from a relatively obscure language used by a handful of programmers at universities, to a "must know" language throughout the computer industry. C has become known as a language that is powerful, fast, and efficient.

This chapter briefly introduces C. It is in two parts. Part 1 describes what a programming language is, and gives the history of the C programming language. This section also introduces some concepts basic to C, such as *structured programming*, *pointer*, and *operator*. Part 2 walks through a C programming session. It emphasizes how a C programmer deals with a real problem, and demonstrates some of the aspects of the language.

This chapter is not designed to teach you the entire C language. It will introduce you to C, so you can read the rest of this manual with some understanding. We urge you to look up individual topics of C programming in the Lexicon, and especially to study the example programs given there.

Programming languages and C

Before beginning with C, it is worthwhile to review how a microprocessor and a computer language work.

A *microprocessor* is the part of your computer that actually computes. Built into it is a group of *instructions*. Each instruction tells the microprocessor to perform a task; for example, one instruction adds two numbers together, another stores the result of an arithmetic operation in memory, and a third copies data from one point in memory to another.

Together, a microprocessor's instructions form its *instruction set*. The instruction set is, in effect, the microprocessor's "native language".

A microprocessor also contains areas of very fast storage, called *registers*. The registers are essential to arithmetic and data handling within the microprocessor. How many registers a microprocessor has, and how they are designed, help to determine how much memory the microprocessor can read and write, or *address*, and how the microprocessor handles data.

A *computer language*, as the name implies, lets a human being use the microprocessor's instruction set. The lowest level language is called "assembly language". In assembly language, the programmer calls instructions directly from the microcomputer's instruction set, and manipulates the registers within the microprocessor. To write programs in assembly language, a programmer must know both the microprocessor's instruction set and the configuration of its registers.

Assembly and high-level languages

With assembly language, the programmer can tailor the program specifically to the microprocessor. However, because each microprocessor has a unique instruction set and configuration of registers, a program written in one microprocessor's assembly language cannot be run on another microprocessor. For example, no program written in the assembly language for the Motorola 68000 microprocessor can be run on the IBM PC or any PC-compatible computer. The program must be entirely rewritten in the assembly language for the Intel 8086 microprocessor, which is difficult and time consuming.

A *high-level language* helps programmers to avoid these problems. The programmer does not need to know the microprocessor in detail; instead of specific microprocessor instructions, he writes a set of logical constructions. These constructions are then handed to another program, which translates them into the instructions and registers calls used by a specific microprocessor. In theory, a

program written in a high-level language can be run on any microprocessor for which someone has written a translation program.

A high-level language allows the programmer to concentrate on the task being executed, rather than on the details of registers and instructions. This means that programs can be written more quickly than in assembly language, and can be maintained more easily.

So, what is C?

C was invented in the mid-1970s by Dennis Ritchie, a programmer at Bell Laboratories. Ritchie created C specifically to re-write the UNIX operating system from PDP-11 assembly language. Ritchie designed C to have the power, speed, and flexibility of assembly language, but the portability of high-level languages.

In 1978, Ritchie and Brian W. Kernighan published *The C Programming Language*, which describes and defines the C language. *The C Programming Language* is the "bible" of C, a standard work to which all programmers can refer when writing their programs.

Because C is modeled after assembly language, it has been called a "medium-level" language. The programmer doesn't have to worry about specific registers or specific instructions, but he can use all of the power of the computer almost as directly as he can with assembly language. The price is that a C program often can be terse and difficult to understand.

Also, because C was written by experienced programmers for experienced programmers, it makes little effort to protect a programmer from himself. A programmer can easily write a C program that is legal and compiles correctly but crashes the system. Also, C's punctuation marks, or "operators", closely resemble each other. Thus, a mistake in typing can create a legal program that compiles correctly but behaves very differently from what you expect.

Structured programming

C is a *structured language*. This means that a C program is assembled from a number of sub-programs, or *functions*, each of which performs a discrete task. If this concept is difficult to grasp, consider the following example.

Suppose you want to turn a file of text into upper-case letters and print it on the screen. This job seems simple, but a program to do it must perform five tasks:

1. Accept the name of the file to open.
2. Open the file so it can be read, in much the same way that you must open a book before you can read it.
3. Read the text from the file.
4. Turn what is read into upper-case letters.
5. Finally, print the transformed text onto the screen.

A good program will also perform the following tasks:

1. Check that the file requested actually exists.
2. Check that the file requested is actually a text file rather than a file of binary information; the latter makes very little sense when printed on the screen.
3. Close the program neatly when the work is finished.
4. Stop processing and print an error message if a problem occurs.

Let's C

A structured language like C allows you to write a separate function for each of these tasks.

A structured programming language offers two major advantages over a non-structured language. First, it is easier to debug a function than an entire program because the function can be unplugged from the program as a whole, made to work correctly, and then plugged back in again. Second, once a function works, it can be used again and again in different programs. This allows you to create *libraries* of reliable functions that you can pull off the shelf whenever you need them.

The functions within a program communicate by passing values to each other. The value being passed can be an integer, a character, or—most commonly—an address within memory where a function can find data to manipulate. This passing of addresses, or *pointers*, is the most efficient way to manipulate data because by receiving one number, a function can find its way to a large amount of data. This speeds up a program's execution.

C adds some extra tools to help you construct programs. To begin, C allows you to store functions in compiled form. These precompiled functions are added only when the program is finally loaded into memory; this spares you the trouble of having to recompile the same code again and again. Second, C adds a preprocessor that expands definitions, or *macros*, and pulls in special material stored in *header files*. This allows you to store often-used definitions in one file and use them just by adding one line to your program.

Compiling a C program

When **Let's C** compiles a C program, it invokes a number of sub-programs, or *phases*, each of which performs part of the work of turning your file of C code into an executable program. The phases are as follows:

cpp The preprocessor. This reads the file of source code, adds any header files that you have requested, and expands any user-defined macros in the program.

cc0 The preprocessed file is then handed to **cc0**, the parser, which examines the program to see that it is written in legal C and translates it into a logical structure, or *tree*.

cc1 The output of the parser is then handed to **cc1**, the code generator, which translates the logical structure created by the parser into machine instructions.

cc2 The output of the code generator is then handed to **cc2**, the optimizer, which examines the code, eliminates redundant instructions, and then writes the object module file. The output of **cc2** is the relocatable object module, which always has the suffix **.obj**.

The relocatable object module is handed to MS-LINK, the linker, which opens the libraries and adds the library functions to create the executable program. What the linker does will be explained in more detail below.

This sounds complicated, and it is; for that reason, **Let's C** includes a command, called **cc**, that guides a program through the compilation process automatically. For example, to compile the program **test.c** with **Let's C**, all you have to do is type:

```
cc test.c
```

or use the MWS display interface, as described in section 1, *Tutorial Introduction*. **cc** takes care of the rest.

Writing a C program

As noted above, a C program consists of a bundle of sub-programs, or *functions*, which link together to perform the task you want done. Every C program must have at least one function that is called **main**. This is the main function; when the computer reads this, it knows that it must begin to execute the program. All other functions are subordinate to **main**. When the **main** function is finished, the program is over.

Here is a simple C program; all it does is print the message "Hello, world!" on the screen:

```
main()
{
        printf("Hello, world!\n");
}
```

As you can see, this program begins with the word **main**. The program begins to work at this point. The parentheses after **main** enclose all of the *arguments* to **main** — or would, if this program's **main** took any. An argument is an item of information that a function uses in its work.

The braces '{' and '}' enclose all the material that is subsidiary to **main**.

The word "printf" *calls* a function called **printf**. This function performs formatted printing. The line of characters (or "string") *Hello, world!* is the argument to **printf**: this argument is what **printf** is to print.

The characters '\n' stand for a carriage return; this ensures that when the program is finished, the cursor is not left fixed in the middle of the screen. Finally, the semicolon ';' at the end of the command indicates that the command is finished.

One point to remember is that **printf** is *not* part of the C language. Rather, it is a *function* that was written by Mark Williams Company, then compiled and stored in a library for later use. This means that you do not have to re-invent a formatted printing function to perform this simple task: all you have to do is *call* the one that Mark Williams Company has written for you.

Although most C programs are more complicated than this example, every C program has the same elements: a function called **main**, which marks where exection begins and ends; braces that fence off blocks of code; functions that are called from libraries; and data passed to functions in the form of arguments.

A sample C programming session

This section walks you through a C programming session. It shows how you can go about planning and writing a program in C.

C allows you to be precise in your programming, which should make you a stronger programmer. Be careful, however, because C does exactly what you tell it to do: if you make a mistake, you can produce a legal C program that does very unexpected things.

Designing a program

Most programmers prefer to work on a program that does something fun or useful. Therefore, we will write something useful: a version of the UNIX utility **more**. It will do the following:

1. Open a text file on disk.
2. Display its contents in 23-line portions (one full screen).
3. After a portion is displayed, wait to see if the user wants to see another portion. If the user presses the space bar, display another portion; if he types anything else, exit.
4. Exit automatically when the end of file is reached.

As you can see, the first step in writing a program is to write down what the program is to do, in as much detail as you can manage, and in complete sentences.

Now, invoke the MicroEMACS editor and get ready to type in the program. Use the command

```
me more.c
```

or use the MWS display interface as described in section 1 of this manual. Note that the suffix **.c** on the file name indicates that this is a file of C code. If you do not use this suffix, **Let's C** will not

recognize that this is a file of C code, and will refuse to compile it.

Begin by inserting a description of the program into the top of the file in the form of a *comment*. When a C compiler sees the symbol '/*', it throws away everything it reads until it sees the symbol '*/'. This lets you insert text into your program to explain what the program does.

Now, type the following:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */
```

Save what you have typed by pressing **<ctrl-X>** and then **<ctrl-S>**. Now, anyone, including you, who looks at this program will know exactly what it is meant to do.

The main function

As described earlier, the C language permits *structured programming*. This means that you can break your program into a group of discrete functions, each of which performs one task. Each function can be perfected by itself, and then used again and again when you need to execute its task. C requires, however, that you signal which function is the *main* function, the one that controls the operation of the other functions; thus, each C program must have a function called **main()**.

Now, add **main()** to your program. Type the code that is shaded, below:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

main()
{
}
```

The parentheses "()" show that **main** is a function; if **main** were to take any arguments, they would be named between the parentheses. The braces "{}" delimit all code that is subordinate to **main**; this will be explained in more detail below.

Note that the shortest legal C program is **main(){}**. This program doesn't do anything when you run it, but it will compile correctly and generate an executable file.

Now, try compiling the program. Save your text by typing **<ctrl-X><ctrl-S>**, and then exit from the editor by typing **<ctrl-X><ctrl-C>**. Compile the program by typing:

```
cc more.c
```

or use the MWS display interface, as described in section 1. When compilation is finished, type **more**. MS-DOS pauses for a moment, and then returns the prompt to your screen. As you can see, you now have a legal, compilable C program, but one that does nothing.

Opening a file and showing text

The next step is to install routines that open a file and print its contents. For the moment, the program will read only a file called **tester**, and not break it into 23-line portions.

Type the shaded lines into your program, as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>

main()
{
        char string[128];
        FILE *fileptr;

/* Open file */
        fileptr = fopen("tester", "r");

/* Read material and display it */
        for (;;)
        {
                fgets(string, 128, fileptr);
                printf("%s\n", string);
        }
}
```

Note first how comments are inserted into the text, to guide the reader.

Now, note the lines

```
        char string[128];
        FILE *fileptr;
```

These *declare* two data structures. That is, they tell **Let's C** to set aside a specific amount of memory for them.

The first declaration, **char string[128];**, declares an array of 128 **char**s. A **char** is a data entity that is exactly one byte long; this is enough space to store exactly one alphanumeric character in memory, hence its name. An *array* is a set of data elements that are recorded together in memory. In this instance, the declaration sets aside 128 **char**s-worth of memory. This declaration reserves space in memory to hold the data that your program reads.

The second declaration, **FILE *fileptr**, declares a *pointer* to a **FILE** structure. The asterisk shows that the data element points to something, rather than being the thing itself. When a variable is declared to be a pointer, **Let's C** sets aside enough space in memory to hold an *address*. When your program reads that address, it then knows where the actual data are residing, and looks for them there. C uses pointers extensively, because it is much more efficient to pass the address of data than to pass the data themselves. You may find the concept of pointers to be a little difficult to grasp; however, as you gain experience with C, you will find that they become easy to use.

The **FILE** structure is the data entity that holds all the information your program needs to read information from or write information to a file on the disk. For now, all you need to remember is that this declaration sets aside a place to hold a pointer to such a structure, and the structure itself holds all of the information your program needs to manipulate a file on disk. In effect, the variable **fileptr** is used within your program as a synonym for the file itself.

Now, the line

```
fileptr = fopen("tester", "r");
```

opens the file to be read. The function **fopen** *opens* the file, fills the **FILE** structure, and fills the variable **fileptr** with the address of where that structure resides in memory.

fopen takes two arguments. The first is the name of the file to be opened, within quotation marks. The second argument indicates the *mode* in which to open the file; **r** indicates that the file will be read only.

The lines

```
for(;;)
{
```

begin a *loop*. A loop is a section of code that is executed repeatedly until a condition that you set is met. For example, you may define a loop that executes until the value of a particular variable becomes greater than zero.

for is built into the C language. Note that it has braces, just like **main()** does; these braces mean that the following lines, up to the next right brace (}) are part of this loop. You can set conditions that control how a **for** loop operates; in its present form, it will loop forever. This will be explained in more detail shortly.

Two library functions are executed within the loop. The first,

```
fgets(string, 128, fileptr);
```

reads a line from the file named in the **fileptr** variable, and writes it into the character array called **string**. The middle argument ensures that no more than 128 characters will be read at a time. The second line within this loop,

```
printf("%s\n", string);
```

prints out the line. **printf** is a powerful and subtle function; in its present form, it prints on the screen the string named in the variable *string*.

Finally, the line at the top of the program

```
#include <stdio.h>
```

tells **Let's C** to read a *header file* called **stdio.h**. The term "STDIO" stands for "standard input and output"; **stdio.h** declares and defines a number of routines that will be used to read data from a file and write them onto the screen.

When you have finished typing in this code, again compile the program as you did earlier. If an error occurs, check what you have typed and make sure that it *exactly* matches the code shown on the previous page. If you find any errors, fix them and then recompile. If errors persist, see the sections *Error Messages* and *Questions and Answers* for help.

When compilation is finished, execute **more** as you did earlier. The file **tester** is included with **Let's C**. You will see the text from **tester** scroll across the screen. When the text is finished, however, the DOS prompt does not return; you have not yet inserted code that tells the program to recognize that the file is finished. Type **<ctrl-C>** to break the program and return to DOS.

Accepting file names

Of course, you will want **more** to be able to display the contents of any file, not just files named **tester**. The next step is to add code that lets you pass arguments to the program through its command line. This task requires that you give the **main()** function two arguments; by tradition, these are always called **argc** and **argv**. How they work will be described in a moment.

The enhanced program now appears as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
/* Declare arguments to main() */
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;

/* Open file */
        fileptr = fopen(argv[1], "r");

/* Read material and display it */
        for (;;)
        {
                fgets(string, MAXCHAR, fileptr);
                printf("%s\n", string);
        }
}
```

First, a small change has been added: the line

```
        #define MAXCHAR 128
```

defines the *manifest constant* **MAXCHAR** to be equivalent to 128. This is done because the "magic number" 128 is used throughout the program. If you decide to change the number of characters that this program can handle at once, all you would have to do is to change this one line to alter the entire program. This cuts down on mistakes in altering and updating the program. If you look lower in the program, you will see that the declaration

```
        char string[128]
```

has been changed to read

```
        char string[MAXCHAR]
```

The two forms are equivalent; the only difference is that the latter is easier to use. It is a good idea to use manifest constants wherever possible, to streamline changes to your program.

Now, look at the line that declares **main()**. You will see that **main()** has two arguments: **argc** and **argv**.

The first is an **int**, or integer, as shown by its declaration — **int argc;**. **argc** gives the *number* of entries typed on a command line. For example, when you typed

```
more filename
```

the value of **argc** was set to two: one for the command name itself, and one for the file-name argument. **argc** and its value are set by **Let's C**. You do not have to do anything to ensure that this value is set correctly.

argv, on the other hand, is an array of pointers to the command line's elements. In this instance, **argv[1]** points to name of the file that you want **more** to read. This, too, is set by **Let's C**, and works automatically.

If you look below at the line that delares **fopen()**, you will see that **tester** has been replaced with **argv[1]**; this means that you want **fopen()** to open the file named in the first argument to the **more** command.

Now, try running the program by typing

```
more tester
```

more will open **tester** and display its contents on the screen. You still need to type **<ctrl-C>** when the file is finished; the code to recognize the end of the file will be inserted later.

Also, be sure that you give the command only one file name as an argument, no more and no less. Code that checks against errors has not yet been inserted, and handing it the wrong number of arguments could cause MS-DOS to crash.

Error checking

Obviously, the program runs at this stage, but is still fragile, and could cause problems for you. The next step is to stabilize the program by writing code to check for errors. To do so, a programmer must first write code to capture error conditions, and then write a routine to react appropriately to an error.

Our edited program now appears as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
/* define arguments to main() */
int argc;
char *argv[];
{
       char string[MAXCHAR];
       FILE *fileptr;
```

```
/* Check if right number of arguments was passed */
        if ((argc-1) != 1)
                error("Usage: more filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Read material and display it */
        for (;;)
        {
                fgets(string, MAXCHAR, fileptr);
                printf("%s\n", string);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

The additions to the program are introduced by comments.

The first addition

```
        if ((argc-1) != 1)
                error("Usage:  more filename");
```

checks to see if the correct number of arguments was passed on the command line; that is to say, it checks to make sure that you named a file when you typed the **more** command.

As noted above, **argc** is the number of arguments on the command line, or rather, the number of arguments plus one, because the command name itself is always considered to be an argument. The statement **if((argc-1) != 1)** will check this. The **if** statement is built into C. If the condition defined between its parentheses is true, then do something, but if it is not true, do nothing at all. The operator **!=** means "does not equal". Therefore, our statement means that if **argc** minus one is not equal to one (in other words, if there is not one and only one argument to the **more** command), execute the function **error**. **error** is defined below.

Our **fopen** function also has some error checking added (which will be described in a moment):

```
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");
```

fopen will return a value called "NULL" if, for any reason, it cannot open the file you requested. Thus, our new **if** statement says that if **fopen** cannot open the file named on the first argument to the command line (that is, **argv[1]**), it should invoke the **error** function.

C always executes nested functions from the "inside out". That means that the innermost function (that is, the function that is enclosed most deeply within the pairs of parentheses) is executed first. Its result, or what it *returns*, is then passed to next outermost function as an argument; that function is then executed and what it returns is, in turn, passed to the function that encloses it, and so on. In this instance, the innermost function is

```
        fileptr = fopen(argv[1], "r")
```

fopen is executed and what it returns is written into **fileptr**. What **fopen** returned is then passed to

the next outer operation; in this case, it is compared with NULL, as follows:

```
(fileptr = fopen(argv[1], "r") == NULL
```

What that operation returns is then passed to the outermost function, in this case the **if** statement, which evaluates what it is passed, and acts accordingly. If **fileptr** is NULL (that is, if **fopen** couldn't open the file), the **if** statement will be true and the **error** function will be called. If, however, the file was opened, **fileptr** will not equal NULL and the program will proceed.

As this example shows, C allows a programmer to nest functions quite deeply. Although nested functions are sometimes difficult to untangle when you read them, they make programming much more convenient.

Finally, at the bottom of the file is a new function, called **error**:

```
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

This function stands outside of **main**, as you can tell because it appears outside of **main**'s closing brace. This function is called only when your program needs it. If there are no errors, the program progresses only until the closing brace and the **error** function is never called.

error takes one argument, the message that is to be printed on the screen. This message is defined by the routine that calls **error**. **error** uses the function **printf** to print the message, then calls the **exit** function; this, as its name implies, causes the program to stop. The argument **1** is a special signal that tells MS-DOS that something went wrong with your program.

When the error checking code is inserted, recompile the program without an argument. Previously, this would crash MS-DOS; now, all it does is print the message

```
Usage: more filename
```

and terminate the program.

Print a portion of a file

So far, our utility just opens a file and streams its contents over the screen. Now, you must insert code to print a 23-line portion of the file. At present, it will only print the first 23 lines, and then exit.

To do so, you must insert another **for** loop. Unlike our first loop, which ran forever, this one will cycle only 23 times, and then stop. Our updated program appears as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128
```

```
main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if ((argc-1) != 1)
                error("Usage: more filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines */
        for (;;)
        {
                for (ctr = 0; ctr < 23; ctr++)
                {
                        fgets(string, MAXCHAR, fileptr);
                        printf("%s\n", string);
                }
                exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

The new **for** loop is nested inside the loop governed by **for(;;)**. The program also declares a new variable, **ctr**, at the beginning of the program. **ctr** keeps track of how many times the loop has executed. Now, look at the line:

```
        for (ctr = 0; ctr < 23; ctr++)
```

It has three sub-statements, which are separated by semicolons. The first sub-statement sets **ctr** to zero; the second says that execution is to continue as long as **ctr** is less than 23; and the third says that **ctr** is to be increased by one every time the loop executes (this is indicated by the **++** appended to **ctr**). With each iteration of this loop, **fgets** reads a line from the file named on the **more** command line, and **printf** prints it on the screen.

Also, an **exit** call has been set after this new loop; this ensures that the program will exit automatically after the loop has finished executing. This is a temporary measure, to make sure that you no longer have to type **<ctrl-C>** to return to MS-DOS.

When you have updated the program, recompile it in the usual way. When you run it, **more** will show the first 23 lines of the file, and then the MS-DOS prompt will return.

The program is now approaching its final form.

Checking for the end of file

The next-to-last step in preparing the program is teaching it to recognize the end of a file when it sees it. This does not appear to be needed now because the program exits automatically after 23 lines or fewer, but it will be quite necessary when the program begins to display more than one 23-line portion of text.

The libraries included with **Let's C** include a function that checks for the end of file (or EOF); it is called **feof()**. Before the program attempts to print out a line of text, it should check if the end of the file has been reached. This means placing **feof** in an **if** statement; the statement will take advantage of the fact that **feof** outputs, or *returns*, a zero if the end of file has not been reached, and returns a number other than zero if the end of file has been reached. The **if** statement will capture what **feof** returns, and continue execution as long as the value of the number returned is zero.

The updated program now appears as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if ((argc-1) != 1)
                error("Usage: more filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");
```

```
/* Output 23 lines, while checking for EOF */
        for (;;)
        {
                for (ctr = 0; ctr < 23; ctr++)
                {
                        if (feof(fileptr) == 0)
                        {
                                fgets(string, MAXCHAR, fileptr);
                                printf("%s\n", string);
                        }
                        else
                                exit(0);
                }
                exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

First, note that the comment that describes the program's output has been changed to reflect our changes to the program. It is important for a programmer to ensure that the comments and the code are in step with each other.

Our new **if** statement

```
        if (feof(fileptr) == 0)
        {
```

checks what **feof** returns: if it returns zero, the end of the file has not been reached, the **if** statement is true, and the program prints out the next line. If it returns a number other than zero, the end of file has been reached, the **if** statement is false so the **else** statement is executed, which causes **more** to exit. **feof** takes one argument, which is the **FILE** that was defined by **fopen**.

Note, too, that a new control statement is introduced: **else**. This, like **if**, is built into the C language. An **else** statement is always paired with an **if** statement; together, they mean that if the condition for which **if** is testing is true, the program should do one thing; otherwise, it should do something else. In this case, the program says that if the end of file has not been reached, another line should be read from the file and printed on the screen; however, if it has been reached, then the program should exit. As you can imagine, **if/else** pairs are common in C programming; they are logical and useful.

One more task must be done on our program; then it is finished.

Polling the keyboard

For the program to be complete, it has to ask you if you want to see another 23-line portion of text. The program should write another portion if you press the space bar, and exit if you type anything else.

The program will use a new function, **getcnb**, to accomplish this task. **getcnb** reads what you type in an unbuffered fashion; that means that you do not have to type the carriage return key for the keystroke to be read by the program. This is placed within an **if** statement that compares what character is typed with the space character. If they are not the same (as indicated by the operator **!=**), the program will exit; otherwise, it will loop through again and show another 23 lines.

Let's C

When these changes are inserted, the program is complete:

```
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if ((argc-1) != 1)
                error("Usage: more filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines, while checking for EOF */
        for (;;)
        {
                for (ctr = 0; ctr < 23; ctr++)
                {
                        if (feof(fileptr) == 0)
                        {
                                fgets(string, MAXCHAR, fileptr);
                                printf("%s\n", string);
                        }
                        else
                                exit(0);
                }
/* Read keyboard; exit if not <space> */
                if (getcnb() != ' ')
                        exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

After you have inserted these changes, again compile the program.

When compilation is finished, try typing

```
more more.c
```

The first 23 lines of the source code to the program now appear on your screen. Hit the space bar; the next 23 lines appear. Now, type any other key: the program exits.

You now have a simple but helpful **more** utility.

For more information

This section has given you a brief, concentrated introduction to writing a C program. If you are new to programming, much of what happened must seemed strange, but we hope it helped you to appreciate the logic of how C works.

Numerous books are on the market to teach beginners how to program in C; see the bibliography at the end of section 1 of this manual for a list of them. Also, look at the sample C programs in the Lexicon. These demonstrate how to use many of the functions available to you with **Let's C**.

With patience, you should discover that programming with C is one of the greatest pleasures to be had with a computer: few feats are as satisfying as delving into the machine and having it do exactly what you want it to do.

Where to go from here

The following section, *Advanced compiling*, introduces some of the more sophisticated features of the **Let's C** compiler. You should look through this section when you feel that you are ready for advanced programming.

If you have any questions about any of the features of **Let's C**, or about any of the functions that were described in this tutorial, look in the Lexicon. For example, if you have a question about **feof** or **printf**, look them up in the Lexicon. There, you will full descriptions of how to use them, plus sample C programs that show how to use them. By typing, compiling, and running the sample programs, you will quickly learn how to use the C language.

Compiling with Let's C

This section describes how to compile C programs with **Let's C**.

In brief, a C compiler transforms files of C source code into machine code. Compilation involves several steps; however, **Let's C** simplifies it with the **cc** command, which controls all the actions of the compiler.

The phases of compilation

Let's C is not just one program, but a number of different programs that work together. Each program performs a *phase* of compilation. The following summarizes each phase:

cpp The C preprocessor. This processes any of the '#' directives, such as **#include** or **#ifdef**, and expands macros.

cc0 The parser. This phase parses programs. It translates the program into a parse-tree format, which is independent of both the language of the source code and the microprocessor for which code will be generated.

cc1 The code generator. This phase reads the parse tree generated by **cc0** and translates it into machine code. The code generation is table driven, with entries for each operator and addressing mode.

cc2 The optimizer/object generator. This phase optimizes the generated code and writes the object module.

cc3 **Let's C** also includes a fifth phase, called **cc3**, which can be run after the object generator, **cc2**. **cc3** generates a file of assembly language instead of a relocatable object module. This phase is optional, and allows you to examine the code generated by the compiler. If you want **Let's C** to generate assembly language, use the **-VASM** option on the **cc** command line.

Unless you specify the **-VASM** option, **Let's C** creates an *object module* that is named after the source file being compiled. This module has the suffix **.obj**. An object module is *not* executable; it contains only the code generated by compiling a C source file, plus information needed to link the module with other program modules and with the library functions.

As the final step in its execution, **cc** calls the linker **ld** to produce an executable program.

Edit errors automatically

The first option, and one that you'll use most often, is the MicroEMACS option **-A**. Often when you're writing a new program, you try to compile it, only to have the compiler tell you that you've made a mistake. You must then invoke your editor, change the program, exit from the editor, and start compiling the program again.

To make this process easier, **cc** command has the *automatic* (or MicroEMACS) option, **-A**. If **Let's C** detects any errors in your program, it will automatically invoke the MicroEMACS screen editor. MicroEMACS will display all error messages in one window and your source code in another, with the cursor set at the number of the line where the first error occurred.

Try the following example. Use MicroEMACS to create a program called **error.c**. To invoke MicroEMACS, type the command

```
me error.c
```

at the MS-DOS prompt, or use the display interface to MWS, the Mark Williams shell, as described in section 1 of this manual. Then type the following code:

```
main()
{
        printf("Hello, world")
}
```

Note that the semicolon was left off of the **printf** statement. Type **<ctrl-X><ctrl-S>** to save the file to disk, and **<ctrl-X><ctrl-C>** to exit from MicroEMACS. Now, try compiling **error.c** with the following **cc** command:

```
cc -A error.c
```

or use MWS's display interface, as described in section 1. You will see no messages from the compiler because they are all being diverted into a file to be used by MicroEMACS. Then, MicroEMACS will appear automatically. In the upper window you will see the message:

```
4: missing ';'
```

and in the lower window you will see your source code for **error.c**, with the cursor set on line 4. If you had more than one error, typing **<ctrl-X>>** would move you to the next line with an error in it; typing **<ctrl-X><** would return you to the previous error.

With some errors, such as those for missing braces or semicolons, the compiler cannot always tell exactly which line the error occurred on; it will point to a line that is near the source of the error.

Now, use **<ctrl-E>** to move the cursor to the end of line 3, and type a semicolon to correct the error. Type **<ctrl-X><ctrl-S>** to save the file to disk, and then type **<ctrl-X><ctrl-C>** to exit from MicroEMACS. **cc** will recompile the program automatically, to produce a normal working executable file.

cc will continue to invoke the MicroEMACS editor either until the program compiles without error, or until you exit from the editor by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

Renaming executable files

When **Let's C** compiles a source file, by default it names the executable program after the source file. For example, when you compiled **error.c**, **Let's C** automatically named the executable file **error.exe**.

If you wish, you can give the executable file a different name. Use the **-o** (output) option, followed by the desired name. For example, should you wish the executable file to have the name **example.exe**, use the command:

```
cc -o example.exe error.c
```

This command will compile the source file **error.c** and generate an executable file called **example.exe**. The suffix **.exe** tells MS-DOS that the file is executable.

Floating-point numbers

Often, you will need to use floating-point numbers in your programs. If you are unsure what a floating-point number is, see the Lexicon entry for **float**.

The routines that print floating-point numbers are large, and most C programs do not need to print floating-point numbers; therefore, the code to perform floating-point arithmetic is not included in a program by default. You must ask **Let's C** to include these routines with your program by using the **-f** option with the **cc** command.

For example, if the program **example.c** used floating-point numbers, you would compile it with the following command line:

```
cc -f example.c
```

If your program prints floating-point numbers or reads them from an input device, and it is *not* compiled with the **-f** option, it will print the following error message when it is run:

```
You must compile with the -f option
to include printf() floating point!
```

Compiling multiple source files

Many programs are built from more than one file of C source code. For example, the program **factor**, which is provided with **Let's C**, is built from the C source files **factor.c** and **atod.c**. To produce the executable program **factor**, both source files must be compiled; the linker **ld** then joins them to form an executable file.

To compile a program that uses more than one source file, type all of the source files onto the **cc** command line. For example, to compile **factor** type the following:

```
cc -f factor.c atod.c -lm
```

This command compiles both C source files to create the program **factor**.

When the **cc** command line includes several file name arguments, by default it uses the *first* to name the executable file. In the above example, **cc** produces the non-executable object modules **factor.obj** and **atod.obj**, and then links them together to produce the executable file **factor.exe**.

The argument **-lm** tells **cc** to include routines from the mathematics library when the object modules are linked. This option must come *after* the names of all of the source files, or the program will not be linked correctly.

Wildcards

A *wildcard* character is one that represents a variety of characters. MS-DOS recognizes the asterisk '*' and the question mark '?'. The asterisk can represent any string of characters of any length (including no character at all), whereas the question mark can represent any one character.

For example, if the current directory held the following files:

```
a.c
ab.c
abc.c
abcd.c
```

typing **dir a?.c** would print:

```
ab.c
```

whereas typing **dir a*.c** would print all four files.

The **cc** command lets you use wildcards in your command line to save you time and effort. For example, you can compile all of the C source files in the current directory simply by typing:

```
cc *.c
```

This command compiles all of the files with the suffix **.c** and links the resulting object modules.

In another example, if the program **example** were built from the source files **example1.c**, **example2.c**, and **example3.c**, you could compile them with the following command:

```
cc example?.c
```

Tailoring the command line interface

With **Let's C**, you can tailor the command-line interface that your compiled programs use. Some programs do not use command-line arguments; others take a few; whereas others may need to read the environment and expand wildcard characters. The following options allow you to select the interface you want for your program.

The option **-na** (for "no arguments") tells **Let's C** that a program does not use command line arguments. The **-na** option may be used with or without the **-ns** option, which suppresses STDIO.

The option **-w** (for "wildcard") tells **Let's C** to include code that expands the wildcards '?' and '*' used in command-line arguments. For example, if the program **example.exe** is compiled with the **-w** option, it will expand the command:

```
example *.c
```

The wildcard argument ***.c** will expand into all file names in the current directory that end in **.c**.

If your program defines a global array **char _cmdname[]** that gives the name of the command, then compiling the program with the **-w** option will include code that fills in **argv[0]** with the command name and looks for environmental variables of the form *name***HEAD** and *name***TAIL**. If found, these are added to the **argv[]** array, respectively, before and after the command-line arguments.

For example, the word-count command **wc** is built with the **-w** option. If you set the environmental variable **WCHEAD** to **-l**, then the command

```
wc foo.c
```

has the same effect as the command

```
wc -l foo.c
```

The arguments to the function **main** are usually defined as

```
main(argc, argv)
int argc; char *argv[];
```

On some systems, a third argument is available:

```
main(argc, argv, envp)
int argc;
char *argv[], *envp[];
```

The argument **envp** is a NULL-terminated array of pointers to environmental variables, each of the form *var*=*value*. If a program is compiled without the **-w** option, **Let's C** passes an empty list as **envp**. If a program is compiled with this option, **Let's C** passes an **envp** that points to all of the MS-DOS environmental variables. Note that your program does not have to use **envp**; like **argc** and **argv**, it is available should you want it.

Linking without compiling

When you are writing a program that consists of several source files, you will need to compile the program, test it, and then change one or more of the source files. Rather than recompile all of the source files, you can save time by recompiling only the modified files and relinking the program.

For example, if you modify the **factor** program by changing the source file **factor.c**, you can recompile **factor.c** *and* relink the entire program with the following command:

```
cc -f factor.c atod.obj -lm
```

The first two arguments are the C source file **factor.c** and the *object module* **atod.obj**. **cc** recognizes that **atod.obj** is an object module and simply passes it to the linker **ld** without compiling it. You will find this particularly useful when your programs consist of many source files and you need to compile only a few of them.

To simplify compiling, especially if you are developing systems that use many source modules, you should consider using the **make** command that is included with **Let's C**. For more information on **make**, see the entry in the Lexicon, or see the tutorial for **make** that appears later in this manual.

Compiling without linking

At times, you will need to compile a source file but not link the resulting object module to the other object modules. You will do this, for example, to compile a module that you wish to insert into a library. Use the **-c** option to tell **cc** not to link the compiled program. This option is used most often to create relocatable object modules that can be archived into a library for later use.

For example, if you wanted just to compile **factor.c** without linking it, you would type:

```
cc -c factor.c
```

To link the resulting object module with the object module **atod.obj** and with the appropriate libraries, type the following command:

```
cc -f factor.obj atod.obj -lm
```

Mini-make option

When you write a program that consists of several files of source code, you may find that, at one time or another, you need to alter the code in just one or two files, to update the program or to fix a bug. You must then recompile and relink the program to create an executable file; however, it is wasteful to recompile every file of source code when did not modify all of them. What you need is an easy way to recompile only the files that you edited, and then relink all of the object modules into an executable file.

The **-m** (mini-**make**) option allows you to create an up-to-date version of your program without recompiling all of your source files. When you use the **-m** option, the compiler compares the **date** the source file was last modified with the date its object module was last created. If the object module has a later date than the source file, then the source file has not been modified since it was last compiled, and **Let's C** will not recompile it. It will, however, re-link the previously compiled object module to build a new executable file.

This option is quite useful when recompiling programs that are built out of many different modules because unchanged source files are not recompiled unnecessarily. Note, however, that the **-m** option does not recognize header file dependencies, so you should use it with some caution.

Note, too, that this option will not work properly if you do not reset your system's time whenever you reboot. If you do not, files will be date-stamped to the default time, and **cc** will not be able organize them properly.

Assembly-language files

C makes most assembly language programming unnecessary. However, you may wish to write small parts of your programs in assembly language for greater speed or to access processor features that C cannot use directly. **Let's C** includes an assembler, named **as**, which is described in detail in the Lexicon.

To compile a program that consists of the C source file **example.c** and the assembly-language source file **example.s**, simply use the **cc** command as usual:

```
cc example1.c example2.s
```

cc recognizes that the suffix **.s** indicates an assembly-language source file, and assembles it with **as**; then it links both object modules to produce an executable file.

If you wish, you can also write programs that combine assembly language with C preprocessor instructions. These files should have the suffix **.m**. When you name a **.m** file in a **cc** command, **cc** will pass it first to the C preprocessor **cpp**, and then pass what **cpp** produces to the assembler **as**. These allow you to write assembly-language programs that are independent of i8086 memory model. For more information on how to use the **.m** format, see the Lexicon entries for **larges.h** and for **as**.

Changing the size of the stack

The *stack* is the segment of memory that holds function arguments, local variables, and function return addresses. **Let's C** by default sets the size of the stack to two kilobytes (2,048 bytes). This is enough stack space for most programs; however, some programs, such as the example program on page 26 of the first edition of *The C Programming Language,* ed. 2, require more than two kilobytes of stack. A program that uses more than its allotted amount of stack will cause a *stack overflow*; this may force you to reboot your computer.

The size of the stack cannot be altered while a program is running. Should your program need more than two kilobytes of stack, use the **-ys** option to the **cc** command. For example, to increase the stack size to 8,000 bytes, use the following command to the **cc** command:

```
cc -ys8000 hello.c
```

Note that this option indicates the number of bytes to which you wish to set the stack, not the number of kilobytes. This must be a decimal number.

i8086 memory models

The i8086/88 microprocessor uses a *segmented architecture*. This means the i8086/88 divides memory into segments of 64 kilobytes each. No program or data element can exceed that limit.

Intel Corporation has devised a number of *models* for organizing the segments of memory into a program that is larger than any single segment. **Let's C** implements the two most useful of these: SMALL model and LARGE model.

SMALL model C programs use 16-bit pointers and **near** calls. Because a 16-bit pointer can address 65,536 bytes (64 kilobytes) of memory, SMALL model programs are limited to 64 kilobytes (one segment) of code and 64 kilobytes of data and stack.

LARGE model C programs use 32-bit pointers and **far** calls. In the LARGE model, the 32-bit pointers are converted by the processor to 20-bit addresses, so LARGE model programs can access up to a total of 1,048,576 bytes (one megabyte) of code and data. The IBM PC and and its imitators have a physical limit of 640 kilobytes.

In terms of execution, LARGE-model programs run more slowly than SMALL-model programs, but for many purposes the advantages of the expanded address space of the LARGE model outweigh the decreased efficiency.

When **Let's C** compiles a program with the **-VSMALL** option, the resulting object module follows the rules of the SMALL model. This is the default setting for the compiler. When the **-VLARGE** option is used with the **cc** command, the object program follows the rules of the LARGE model.

When you compile a program with the **-VLARGE** option, **cc** defines the manifest constant **LARGE** to the C preprocessor. This allows you to use the **#ifdef LARGE** conditional to flag model-dependent code.

Note that you cannot mix SMALL-model object modules with those compiled into LARGE model.

Debugging information

One powerful feature of **Let's C** is its ability to generate programs that you can debug with **csd**, the revolutionary Mark Williams C source debugger. **csd** lets you debug C *source code*: you can use it even if you do not know i8086 assembly language.

csd uses debugging information that **Let's C** writes into the object module as it compiles a C program. Because this information slightly enlarges the file that contains the object modules, **Let's C** does not produce it unless you request it. To include debugging information in an object module, use the **-VCSD** option before the file name argument on the **cc** command line:

```
cc -VCSD hello.c
```

The manual for **csd** describes the C source debugger in full.

A module compiled with the **-VCSD** option will run exactly the same as one compiled without it, but the size of the object module will increase by a few bytes. The size of the executable file will increase, due to the special symbol table that the **-VCSD** option builds.

With some programs that already approach the limits of the SMALL model, compiling with the **-VCSD** option may make them too large to be executed as SMALL model programs. In that case, recompile the program with the **-VCSD** and **-VLARGE** options; the latter option will create a LARGE model output.

To remove the debug symbol table from the programs that you compile with the **-VCSD** option, use the **strip** command. **strip** is described in the Lexicon.

i8087 programs

The Intel i8087 chip is a numeric data processor that is designed to execute mathematics routines. It increases the speed with which programs can compute floating-point numbers. Because of its expense, however, many personal computers do not include this chip.

Let's C by default uses a special set of libraries that *sense* if an i8087 is present. When you compile a program with these libraries and then run it, the library routines automatically check to see if an i8087 is present on your computer. If an i8087 is present, then floating-point arithmetic is automatically computed it; otherwise, it is computed in software. Thus, a program compiled with **Let's C** can be run to best advantage on machines that have an i8087 as well as on machines that do not, without needing to recompile the program.

If you know that the program you are compiling will always be run on a machine with an i8087, you may wish to use the libraries that use the i8087 exclusively. You can do this by specifying the **-VNDP** option to the **cc** command. For example, to compile the program **factor** to run exclusively with an i8087, use the following command:

```
cc -VNDP factor.c atod.c -lm
```

This program will *not* run on a machine that does not have an i8087; however, the executable file will be somewhat smaller than one that uses the sensing libraries, and will run slightly faster.

Options passed to MS-LINK

The compiler controller **cc** passes a number of its options directly to MS-LINK. The following summarizes them.

-y/*switch*
: This option sends *switch* directly to MS-LINK. *switch* can be any MS-LINK command or option.

-ym Tell MS-LINK to create a *map file* that can be used with the MS-DOS utility **DEBUG**. For more information on **DEBUG** and its uses, see your MS-DOS manual.

-yn **Increase the number of segments allowed in a program to 1,024 using the MS-LINK** *segments* switch. Note that the *segments* switch is used only version of MS-LINK later 3.0. Earlier versions use the *x* switch to increase the number of segments.

-ys*number*
Set the stack size to *number* where *number* is a decimal integer that gives the number of bytes you desire. The stack is set by default to two kilobytes; to set the stack, for example, to 16,000 bytes type:

```
cc -ys16000 foo.c
```

-yf Tell MS-LINK to write a linker command file. This option is useful, should you ever have trouble linking a program and wish to see just what MS-LINK is doing, or if you wish to fine-tune how your program is linked.

-yu*name*
Undefine the variable *name* for MS-LINK. This tells MS-LINK to link in the library module called *name* even though it is not named explicitly in your program. For example, the command line

```
cc -yuprintf example.c
```

tells MS-LINK to link the library module **printf** into your program, even if your program does not explicitly call **printf**. This tactic is sometimes quite useful.

Compiling programs without STDIO

STDIO is an abbreviation for *standard input and output*. Library routines use STDIO to write to the screen or read the keyboard. Most of the runtime startup routines included with **Let's C** call STDIO, whether your program uses any STDIO functions or not.

If you have a small program that does not use any of the STDIO functions, you can stop STDIO from being linked into your program by using the **-ns** option. This will make your program noticeably smaller and more efficient. Note that the **-ns** option gives your program a different version of the **exit** command, one that does not call **fclose** or **fflush**

Using default options

To make using **Let's C** even simpler, **cc** helps you specify default options with the environment variables **CCHEAD** and **CCTAIL** These variables give options that **cc** adds to the command line you give it: it adds **CCHEAD** to the start of the command line (after the "cc"), and it appends **CCTAIL** to the end of the command line.

How you can build a *name***HEAD** and *name***TAIL** feature into your program is described above, in the sub-section *Tailoring the command line interface*.

When you installed **Let's C**, the **install** utility instructed you to set **CCHEAD** so that **Let's C** would read the file **CCARGS**. If you wish, though, you can attach additional variables to **CCHEAD**, or add them to the file **ccargs**.

For example, suppose you always wish to use the options **-V** and **-f** (for "verbose" compilation and floating-point routines), and always link in the mathematics library with the **-lm** option (which, as you recall, must be mentioned *after* the source and object modules). Rather than retype these options every time you type a **cc** command line, you can set **CCHEAD** and **CCTAIL** as follows:

```
set CCHEAD=@a:\lib\ccargs\ -V -f
set CCTAIL=-lm
```

Note that if your computer has a hard disk, **CCHEAD** should indicate that **ccargs** is on drive C, rather than drive A, as shown above. Thereafter, when you type

```
cc factor.c atod.c
```

it will be as if you had typed

```
cc -V -f factor.c atod.c -lm
```

in addition to the arguments contained in **ccargs**. These environmental variables allow you to pass variables to **Let's C** with ease. To ensure that these variables are set every time you boot your system, be sure to enter the **set** commands described above into the file **autoexec.bat** on your MS-DOS boot disk.

Where to go from here

For more information on compiling, see the Lexicon entry for **cc**. This entry summarizes all of **cc**'s options, and presents many that are not discussed here. For more information on the assembler **as**, see its entry in the Lexicon as well.

The following section introduces the MicroEMACS screen editor. If you have worked the exercises in this part of the book, you have already used MicroEMACS a little; this tutorial, however, will show you how to use all of its advanced features to input text quickly and easily.

Then comes an introduction to **make**, the Mark Williams programming discipline. If you are building programs that use multiple files of source code, you will find **make** to be an invaluable tool.

Section 6, *Questions and Answers*, answers frequently asked questions about **Let's C** and its utilities. If you have a question about **Let's C**, look here first. You may well find the information you need.

Introduction to MicroEMACS

This section introduces MicroEMACS, the interactive screen editor for **Let's C**. It is written for two types of reader: the one who has never used a screen editor and needs a full introduction to the subject, and the one who has used a screen editor before but wishes to review specific topics.

What is MicroEMACS?

MicroEMACS is an interactive screen editor. An *editor* lets you type text into your computer, name it, store it, and recall it later for editing. **Interactive** means that MicroEMACS will accept an editing command, execute it, display the results for you immediately, then wait for your next command. **Screen** means that you can use nearly the entire screen of your terminal as a writing surface: you can move your cursor up, down, and around your screen to create or change text, much as you move your pen up, down, and around a piece of paper.

These features, plus the others that will be described in the course of this tutorial, make MicroEMACS a tool that is powerful yet easy to use. You can use MicroEMACS to create or change computer programs or any type of text file.

The MS-DOS version of MicroEMACS was adapted by Mark Williams Company from a public-domain program written by David G. Conroy. This tutorial is based on the descriptions in his essay *MicroEMACS: Reasonable Display Editing in Little Computers* MicroEMACS is derived from the mainframe display editor EMACS, which was created at the Massachusetts Institute of Technology by Richard Stallman.

For a summary of MicroEMACS and its commands, see the entry for **me** in the Lexicon.

Keystrokes — <ctrl>, <esc>

The MicroEMACS commands use *control* characters and *meta* characters. Control characters use the **control** key, which is marked *Control* on your keyboard; meta characters use the **escape** key, which is marked *Esc*.

Ctrl works like the **shift** key: you hold it down **while** you strike the other key. Here, this will be represented with a hyphen; for example, pressing the control key and the letter 'X' key simultaneously will be shown as follows:

```
<ctrl-X>
```

The *esc* key, on the other hand, works like an ordinary character. You should strike it first, **then** strike the letter character you want. **Escape** character codes will not be represented with a hyphen; for example, *escape X* will be represented as:

```
<esc>X
```

Becoming acquainted with MicroEMACS

Now you are ready for a few simple exercises that will help you get a feel for how MicroEMACS works.

To begin, type the following command to MS-DOS:

```
me sample
```

If you are using **Let's C** through the MWS display interface, return to the main menu and then press **<return>**. When the **Edit** menu appears, press the **<>** key until the cursor bar is at **New File**; then press **<return>** and type **sample**.

Within a few seconds, your screen will have been cleared of writing, the cursor will be positioned in the upper left-hand corner of the screen, and a command line will appear at the bottom of your screen.

Now type the following text. If you make a mistake, just backspace over it and retype the text. Press the carriage return or enter key after each line:

```
main()
{
        printf("Hello, world!\n");
}
```

Notice how the text appeared on the screen character by character as you typed it, much as it would appear on a piece of paper if you were using a typewriter.

Now, type *<ctrl-X><ctrl-S>*; that is, type *<ctrl-X>*, and then type *<ctrl-S>*. It does not matter whether you type capital or lower-case letters. Notice that this message has appeared at the bottom of your screen:

```
[Wrote 4 lines]
```

This command has permanently stored, or **saved**, what you typed into a file named **sample**.

Type the next few commands, which demonstrate some of the tasks that MicroEMACS can perform for you. These commands will be explained in full in the sections that follow; for now, try them to get a feel for how MicroEMACS works.

Type *<esc><*. Be sure that you type a less-than symbol '<', instead of a comma. Notice that the cursor has returned to the upper left-hand corner of the screen. Type *<esc>F*. The cursor has jumped forward by one word, and is now on the left parenthesis. Type *<ctrl-N>*. Notice that the cursor has jumped to the next line, and is now just to the right of the left brace '{'. Type *<ctrl-A>*. The cursor has jumped to the **beginning** of the second line of your text. Type **<ctrl-N>** again, and the cursor is at the beginning of the third line of the program, the **printf** statement.

Now, type *<ctrl-K>*. The third line of text has disappeared, leaving an empty space. Type *<ctrl-K>* again. The empty space where the third line of text had been has now disappeared.

Type *<esc>>*. Be sure to type a greater-than symbol '>', not a period. The cursor has jumped to the space just below the last line of text. Now type *<ctrl-Y>*. The text that you erased a moment ago has now been restored.

By now, you should be feeling more at ease with typing MicroEMACS's **control** and **escape** codes. The following sections will explain what these commands mean. For now, exit from MicroEMACS by typing *<ctrl-X><ctrl-C>*, and when the message

```
Quit [y/n]?
```

appears type **y** and then **<return>**. This will return you to MS-DOS or MWS.

Beginning a document

If your computer does *not* have a hard disk, do the following before you begin: insert disk 2, the compiler disk, into drive A of your computer. Insert disk 8, which holds the sample programs, into drive B. Then, log into directory **sample** on drive B by typing the following command:

```
cd b:\sample
```

If your system does have a hard disk, log into directory **sample** on your hard disk by typing the following:

```
cd c:\sample
```

Let's C

Now, edit the file called **example1.c**. First, use the **cd** to move to directory **\src**, which is where this file was stored when you installed **Let's C**. If you stored the sample programs in a different directory, then use the **cd** command to transfer to that directory. Now, type the following command:

```
me example1.c
```

If you are working through the MWS display interface, invoke MicroEMACS as follows: First, make sure that you are in the main menu, and that the cursor bar is positioned over **Edit**. Type **<return>**. When the **Edit** menu appear, press the **<>** key to move the cursor bar to **Files**. Press **<return>**. A box will appear on the screen that shows all of the files available for editing. Press the **<>** key until the cursor bar is positioned over the file labelled **example1.c**; then press **<return>**. As you can see, **example1.c** now appears in the command box, which is at the top of the screen. Press **<end>**, to return to the **Edit** menu; then press **<return>**, to execute the command you have just built. This will invoke MicroEMACS to edit the file **example1.c**.

In a moment, the following text will appear on your screen:

```
/*
 * This is a simple C program that computes the results
 * of three different rates of inflation over the
 * span of ten years.  Use this text file to learn
 * how to use MicroEMACS commands
 * to make creating and editing text files quick,
 * efficient and easy.
 */
#include <stdio.h>
main()
{
      int i;                    /* count ten years */
      float w1, w2, w3;         /* three inflated quantities */
      char  *msg = " %2d\t%f %f %f\n";/* printf string */
      i = 0;
      w1 = 1.0;
      w2 = 1.0;
      w3 = 1.0;
      for (i = 1; i<= 10; i++) {
            w1 *= 1.07;         /* apply inflation */
            w2 *= 1.08;
            w3 *= 1.10;
            printf (msg, i, w1, w2, w3);
      }
}
```

When you type the MicroEMACS command and a file name, MicroEMACS *copies* that file into memory. Your cursor also moved to the upper left-hand corner of the screen. At the bottom of the screen appears the *status line*, as follows:

```
-- MicroEMACS -- example1.c -- File: example1.c ----------
```

The word to the left, MicroEMACS, is the name of the editor. The word in the center, *example1.c*, is the name of the **buffer** that you are using. What a buffer is and how it is used will be covered later. The name to the right is the name of the text file that you will be editing.

Moving the Cursor

Now that you have read a text file into memory, you will want to edit it. The first step is to learn to move the cursor.

Try these commands for yourself as they are described in the following paragraphs. That way, you will quickly acquire a feel for handling MicroEMACS's commands. You can also use your **arrow keys** with MicroEMACS. The arrow keys are found on the keypad on the right-hand side of your keyboard. If when you press the arrow keys, numbers appear in the text instead of the cursor being moved, press the **number lock** key, which is the key marked *Num Lock*. That should solve the problem.

Moving the cursor forward

This first set of commands moves the cursor forward.

<ctrl-F>	Move forward one space
<esc>F	Move forward one word
<ctrl-E>	Move to end of line

To see how these commands work, do the following: Type the **forward** command *<ctrl-F>*. This is equivalent to pressing **<**Rationale**>**. As before, it does not matter whether the letter '**F**' is upper case or lower case. The cursor has moved one space to the right, and now is over the character '*' in the first line.

Type *<esc>F*. The cursor has moved one **word** to the right, and is now over the space after the word *this*. MicroEMACS considers only alphanumeric characters when it moves from word to word. Therefore, the cursor moved from under the * to the space after the word **this**, rather than to the space after the *. Now type the **end of line** command *<ctrl-E>*. The cursor has jumped to the end of the line and is now just to the right of the *e* of the word *three*.

Moving the cursor backward

The following summarizes the commands for moving the cursor backwards.

<ctrl-B>	Move back one space
<esc>B	Move back one word
<ctrl-A>	Move to beginning of line

To see how these work, first type the **backward** command *<ctrl-B>*. This is equivalent to pressing **<>**. As you can see, the cursor has moved one space to the left, and now is over the letter *e* of the word *three*. Type *<esc>B*. The cursor has moved one **word** to the left and now is over the *t* in *three*. Type *<esc>B* again, and the cursor will be positioned on the *o* of the word *of*.

Type the **beginning of line** command *<ctrl-A>*. The cursor jumps to the beginnning of the line, and once again is resting over the '/' character in the first line.

From line to line

<ctrl-P>	Move to previous line
<ctrl-N>	Move to next line

These two commands move the cursor up and down the screen. Type the **next line** command *<ctrl-N>*. The cursor jumps to the space before the '*' in the next line. Type the **end of line** command *<ctrl-E>*, and the cursor moves to the end of the second line to the right of the period.

Continue to type *<ctrl-N>* until the cursor reaches the bottom of the screen. This is the same as if you typed **<>**. As you reached the first line in your text, the cursor jumped from its position at the right of the period on the second line to just right of the brace on the last line of the file. When you move your cursor up or down the screen, MicroEMACS will try to keep it at the same position within each line. If the line to which you are moving the cursor is not long enough to have a character at that position, MicroEMACS will move the cursor to the end of the line.

Let's C

Now, practice moving the cursor back up the screen. Type the **previous line** command *<ctrl-P>*. This has the same effect as pressing **<>**. When the cursor jumped to the previous line, it retained its position at the end of the line. MicroEMACS remembers the cursor's position on the line, and returns the cursor there when it jumps to a line long enough to have a character in that position.

Continue pressing *<ctrl-P>*. The cursor will move up the screen until it reaches the top of your text.

Moving up and down by a screenful of text

The next two cursor movement commands allow you to roll forward or backwards by one screenful of text.

<ctrl-V>	Move forward one screen
<esc>V	Move back one screen

If you are editing a file with MicroEMACS that is too big to be displayed on your screen all at once, MicroEMACS will display the file in screen-sized portions (22 lines at a time). The **view** commands *<ctrl-V>* and *<esc>V* allow you to roll up or down one screenful of text at a time.

Type *<ctrl-V>*. Your screen now contains only the last three lines of the file. This is because you have rolled forward by the equivalent of one screenful of text, or 22 lines.

Now, type *<esc>V*. Notice that your text rolls back onto the screen, and your cursor is positioned in the upper left-hand corner of the screen, over the character '/' in the first line.

Moving to beginning or end of text

Finally, these two cursor movement commands allow you to jump immediately to the beginning or end of your text.

<esc><	Move to beginning of text
<esc>>	Move to end of text

The **end of text** command *<esc>>* moves the cursor to the end of your text. Type *<esc>>*. Be sure to type a greater-than symbol '>'; this symbol may have been placed anywhere on your keyboard, although on IBM-style keyboards it appears above the period. Your cursor has jumped to the end of your text.

The **beginning of text** command *<esc><* will move the cursor back to the beginning of your text. Type *<esc><*. Be sure to type a less-than symbol '<'; on IBM-style keyboards it appears above the comma. The cursor has jumped back to the upper left-hand corner of your screen.

These commands will move you immediately to the beginning or the end of your text, regardless of whether the text is one page long or 20 pages long.

Saving text and quitting

If you do not wish to continue working at this time, you should **save** your text, and then **quit**.

It is good practice to save your text file every so often while you are working on it; then, if an accident occurs, such as a power failure, you will not lose all of your work. You can save your text with the **save** command *<ctrl-X><ctrl-S>*. Type *<ctrl-X><ctrl-S>*—that is, first type *<ctrl-X>*, then type *<ctrl-S>*. If you had modified this file, the following message would appear:

```
[Wrote 23 lines]
```

The text file would have been saved to your computer's disk. MicroEMACS will send you messages from time to time; the messages enclosed in square brackets '[' ']' are for your information, and do not necessarily mean that something is wrong. To exit from MicroEMACS, type the **quit** command *<ctrl-X><ctrl-C>*. This will return you to MS-DOS or MWS.

Killing and deleting

Now that you know how to move the cursor, you are ready to edit your text.

To return to MicroEMACS, type the command:

```
me example1.c
```

Within a moment, *example1.c* will be restored to your screen.

By now, you probably have noticed that MicroEMACS is always ready to insert material into your text; unless you use the *<ctrl>* or *<esc>* keys, MicroEMACS will assume that whatever you type is meant to be text and will insert it onto your screen where your cursor is positioned.

The simplest way to erase text is simply to position the cursor to the right of the text you want to erase and backspace over it. MicroEMACS, however, also has a set of commands that allow you to erase text easily. These commands, **kill** and **delete**, perform differently; the distinction is important, and will be explained in a moment.

Deleting versus killing

When text is **deleted**, it is erased completely; however, when text is *killed*, it is copied into a temporary storage area in memory. This storage area is overwritten when you move the cursor and then kill additional text. Until then, however, the killed text is saved. This aspect of killing allows you to restore text that you killed accidentally, and it also allows you to move or copy portions of text from one position to another.

MicroEMACS is designed so that when it erases text, it does so beginning at the **left edge** of the cursor. This left edge is called the *current position*.

You should imagine that an invisible vertical bar separates the cursor from the character immediately to its left; as you enter the various kill and delete commands, this vertical bar moves to the right or the left with the cursor, and erases the characters it touches. Therefore, if you wish to erase a word but wish to keep both spaces around it, position your cursor directly *over* the first character of the word and strike *<esc>D*. If you wish to erase a word **and** the space before it, position the cursor at the space before you strike *<esc>D*, so that the invisible vertical bar sweeps away the space at which the cursor is positioned, as well as the word that follows.

Erasing text to the right

The first two commands to be presented erase text to the **right**.

<ctrl-D>	Delete one character to the right
<esc>D	Kill one word to the right

<ctrl-D> deletes one *character* to the right of the current position. *<esc>D* deletes one *word* to the right of the current position.

To try these commands, type the **delete** command *<ctrl-D>*. The character '/' in the first line has been erased, and the rest of the line has shifted one space to the left.

Now, type *<esc>D*. The '*' character and the word *This* have been erased, and the line has shifted six spaces to the left. The cursor is positioned at the **space** before the word *is*. Type *<esc>D* again. The word *is* has vanished along with the **space** that preceded it, and the line has shifted **four** spaces to the left.

<ctrl-D> **deletes** text, but *<esc>D* **kills** text.

Let's C

Erasing text to the left

You can erase text to the *left* with the following commands:

<del>	Delete one character to the left
<ctrl-H>	Delete one character to the left
<esc><del>	Kill one word to the left
<esc><ctrl-H>	Kill one word to the left

To see how to erase text to the left, first type the **end of line** command *<ctrl-E>*; this will move the cursor to the right of the word *three* on the first line of text. Then, type *<del>*. The second *e* of the word *three* has vanished.

Type *<esc><del>*. The rest of the word *three* has disappeared, and the cursor has moved to the second space following the word *of*.

Move the cursor four spaces to the left, so that it is over the letter *o* of the word *of*. Type *<esc><del>*. The word *results* has vanished, along with the space that was immediately to the right of it. As before, these commands erased text beginning immediately to the **left** of the cursor. The *<esc><del>* command can be used to erase words throughout your text.

If you wish to erase a word to the left yet preserve both spaces that are around it, position the cursor at the space immediately to the right of the word and type *<esc><del>*. If you wish to erase a word to the left plus the space that immediately follows it, position the cursor under the first letter of the **next** word and then type *<esc><del>*.

Typing *<del>* **deletes** text, but typing *<esc><del>* **kills** text.

Erasing lines of text

Finally, the following command erases a line of text:

<ctrl-K>	Kill from cursor to end of line

This command erases the line beginning from immediately to the left of the cursor.

To see how this works, move the cursor to the beginning of line 2. Now, strike *<ctrl-K>*. All of line 2 has vanished and been replaced with an empty space. Strike *<ctrl-K>* again. The empty space has vanished, and the cursor is now positioned at the beginning of what used to be line 3, in the space before ** Use*.

As its name implies, the *<ctrl-K>* command **kills** the line of text.

Yanking back (restoring) text

The following command allows you restore material that you have killed:

<ctrl-Y>	Yank back (restore) killed text

Remember that when material is killed, MicroEMACS has temporarily stored it elsewhere. You can return this material to the screen by using the **yank back** command *<ctrl-Y>*. Type *<ctrl-Y>*. All of line 2 has returned; the cursor, however, remains at the beginning of line 3.

Quitting

When you are finished, do not save the text. If you do so, the undamaged copy of the text that you made earlier will be replaced with the present changed copy. Rather, use the **quit** command *<ctrl-X><ctrl-C>*. Type *<ctrl-X><ctrl-C>*. On the bottom of your screen, MicroEMACS will respond:

```
Quit [y/n]?
```

Reply by typing *y* and a carriage return. If you type *n*, MicroEMACS will simply return you to where

you were in the text. MicroEMACS will now return you to MS-DOS.

Block killing and moving text

As noted above, text that is killed is stored temporarily within the computer. Killed text may be yanked back onto your screen, and not necessarily in the spot where it was originally killed. This feature allows you to move text from one position to another.

Moving one line of text

You can kill and move one line of text with the following commands:

<ctrl-K>	Kill text to end of line
<ctrl-Y>	Yank back text

To test these commands, invoke MicroEMACS for the text *example1.c* by typing the following command:

```
me example1.c
```

or use the MWS interface, as you did earlier. When MicroEMACS appears, the cursor will be positioned in the upper left-hand corner of the screen.

To move the first line of text, begin by typing the **kill** command *<ctrl-K>* twice. Now, press *<esc>>* to move the cursor to the bottom of text. Finally, yank back the line by typing *<ctrl-Y>*. The line that reads

```
/* This is a simple C program that computes the results
```

is now at the bottom of your text.

Your cursor has moved to the point on your screen that is **after** the line you yanked back.

Multiple copying of killed text

When text is yanked back onto your screen, it is **not** deleted from within the computer. Rather, it is simply **copied** back onto the screen. This means that killed text can be reinserted into the text more than once. To see how this is done, return to the top of the text by typing *<esc><*. Then type *<ctrl-Y>*. The line you just killed now appears as both the first and last line of the file.

The killed text will not be erased from its temporary storage until you move the cursor and then kill additional text. If you kill several lines or portions of lines in a row, all of the killed text will be stored in the buffer; if you are not careful, you may yank back a jumble of accumulated text.

Kill and move a block of text

If you wish to kill and move more than one line of text at a time, use the following commands:

<ctrl-@>	Set mark
<ctrl-W>	Kill block of text

If you wish to kill a block of text, you can either type the **kill** command *<ctrl-K>* repeatedly to kill the block one line at a time, or you can use the **block kill** command *<ctrl-W>*. To use this command, you must first set a **mark** on the screen, an invisible character that acts as a signal to the computer. The mark is set with the **mark** command *<ctrl-@>*.

Once the mark is set, you must move your cursor to the other end of the block of text you wish to kill, and then strike *<ctrl-W>*. The block of text will be erased, and will be ready to be yanked back elsewhere.

Try this out on *example1.c*. Type *<esc><* to move the cursor to the upper left-hand corner of the screen. Then type the **set mark** command *<ctrl-@>*. By the way, be sure to type '@', not '2'. MicroEMACS will respond with the message

```
[Mark set]
```

at the bottom of your screen. Now, move the cursor down six lines, and type *<ctrl-W>*. Note how the block of text you marked out has disappeared.

Move the cursor to the bottom of your text. Type *<ctrl-Y>*. The killed block of text has now been reinserted.

When you yank back text, be sure to position the cursor at the *exact* point where you want the text to be yanked back. This will ensure that the text will be yanked back in the proper place.

To try this out, move your cursor up six lines. Be careful that the cursor is at the **beginning** of the line. Now, type *<ctrl-Y>* again. The text reappeared **above** where the cursor was positioned, and the cursor has not moved from its position at the beginning of the line — which is not what would have happened had you positioned it in the middle or at the end of a line.

Although the text you are working with has only 23 lines, you can move much larger portions of text using only these three commands. Remember, too, that you can use this technique to duplicate large portions of text at several positions to save yourself considerable time in typing and reduce the number of possible typographical errors.

Capitalization and other tools

The next commands perform a number of useful tasks that will help with your editing. Before you begin this section, destroy the old text on your screen with the **quit** command *<ctrl-X><ctrl-C>*, and read into MicroEMACS a fresh copy of the program, as you did earlier.

Capitalization and lowercasing

The following MicroEMACS commands can automatically capitalize a word (that is, make the first letter of a word upper case), or make an entire word upper case or lower case.

<esc>C	Capitalize a word
<esc>L	Lowercase an entire word
<esc>U	Uppercase an entire word

To try these commands, do the following: First, move the cursor to the letter *d* of the word **different** on line 2. Type the **capitalize** command *<esc>C*. The word is now capitalized, and the cursor is now positioned at the space after the word. Move the cursor forward so that it is over the letter *t* in *rates*. Press *<esc>C* again. The word changes to *raTes*. When you press *<esc>C*, MicroEMACS will capitalize the **first** letter the cursor meets.

MicroEMACS can also change a word to all upper case or all lower case. (There is very little need for a command that will change only the first character of an upper-case word to lower case, so it is not included.)

Type **<esc>B** to move the cursor so that it is again to the left of the word *Different*. It does not matter if the cursor is directly over the *D* or at the space to its left; as you will see, this means that you can capitalize or lowercase a number of words in a row without having to move the cursor.

Type the **uppercase** command *<esc>U*. The word is now spelled *DIFFERENT*, and the cursor has jumped to the space after the word.

Again, move the cursor to the left of the word *DIFFERENT*. Type the **lowercase** command *<esc>L*. The word has changed back to *different*. Now, move the cursor to the space at the beginning of line 3 by typing *<ctrl-N>* then *<ctrl-A>*. Type *<esc>L* once again. The character "*" is not affected by the command, but the letter *U* is now lower case. *<esc>L* not only shifts a word that is all upper case to

lower case: it can also un-capitalize a word.

The **uppercase** and **lowercase** commands stop at the first punctuation mark they meet *after* the first letter they find. This means that, for example, to change the case of a word with an apostrophe in it you must type the appropriate command twice.

Transpose characters

MicroEMACS allows you to reverse the position of two characters, or **transpose** them, with the **transpose** command *<ctrl-T>*.

Type *<ctrl-T>*. The character that is under the cursor has been transposed with the character immediately to its **left**. In this example,

```
* use this
```

in line 3 now appears:

```
* us ethis
```

The space and the letter *e* have been transposed. Type *<ctrl-T>* again. The characters have returned to their original order.

Screen redraw

Occasionally, while you are editing you may interrupt MicroEMACS to invoke another program, such as an electronic calculator or a clock. When you exit from that program, you may find that it has left material on your screen and scrambled it. Although this extraneous material will **not** be recorded into your text, you will need to redraw your screen in order to continue to edit. The **redraw screen** command *<ctrl-L>* will redraw your screen to the way it was before it was scrambled.

Type *<ctrl-L>*. Notice how the screen flickers and the text is rewritten. Had your screen been spoiled by extraneous material, that material would have been erased and the original text rewritten.

The *<ctrl-L>* command also has another use: you can move the line on which the cursor is positioned to the center of the screen. If you have a file that contains more than one screenful of text and you wish to have that particular line in the center of the screen, position the cursor on that line and type **<ctrl-U><ctrl-L>**. Immediately, the screen will be rebuilt with the line you were interested in positioned in the center.

Return indent

<ctrl-J>	Return and indent

You may often be faced with a situation in which, for the sake of programming style, you need many lines of indented text. After every line, you must return, then tab the correct number of times, then type your text. *Block indents* can be a time-consuming typing chore. The MicroEMACS *<ctrl-J>* command makes this task easier. When you type a file that has many lines of indented text, such as a C program, you can save many keystrokes by using the *<ctrl-J>* command. *<ctrl-J>* moves the cursor to the next line on the screen, and positions the cursor at the previous line's level of indentation.

To see how this works, first move the cursor to the line that reads

```
w3 *= 1.10:
```

Press **<ctrl-E>**, to move the cursor to the end of the line. Now, type **<ctrl-J>**.

As you can see, a new line opens up and the cursor is indented the same amount as the previous line. Type

Let's C

```
/* Here is an example of auto-indentation */
```

This line of text begins directly under the previous line.

Word wrap

<ctrl-X>F Set word wrap

Although you have not yet had much opportunity to use it, MicroEMACS will automatically wrap around text that you are typing into your computer. Word wrapping is controlled with the *word wrap* command **<ctrl-X>F**. To see how the word wrap command works, first exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**; then reinvoke MicroEMACS by typing

```
me cucumber
```

or use the MWS display interface, as you did earlier. When MicroEMACS re-appears, type the following text; however, do *not* type any carriage returns:

```
A cucumber should be
well sliced, and dressed
with pepper and vinegar,
and then thrown out, as
good for nothing.
```

When you reached the edge of your screen, a dollar sign was printed and you were allowed to continue typing. MicroEMACS accepted the characters you typed, but it placed them at a location beyond the right edge of your screen.

Now, move to the beginning of the next line and type **<ctrl-U>**. MicroEMACS will reply with the message:

```
Arg: 4
```

Type **30**. The line at the bottom of your screen now appears as follows:

```
Arg: 30
```

(The use of the *argument* command **<ctrl-U>** will be explained in full in a few sections.) Now type the *word-wrap* command **<ctrl-X>F**. MicroEMACS will now say at the bottom of your screen:

```
[Wrap at column 30]
```

This sequence of commands has set the word-wrap function, and told it to wrap to the next line all words that extend beyond the 30th column on your screen.

The *word wrap* feature automatically moves your cursor to the beginning of the next line once you type past a preset border on your screen. When you first enter MicroEMACS, that limit is automatically set at the first column, which in effect means that word wrap has been turned off.

When you type prose for a report or a letter of some sort, you probably will want to set the border at the 65th column, so that the printed text will fit neatly onto a sheet of paper. If you are using MicroEMACS to type in a program, however, you probably will want to leave word wrap off, so you do not accidentally introduce carriage returns into your code.

To test word wrapping, type the above text again, without using the carriage return key. When you finish, it should appear as follows:

```
A cucumber should be well
sliced, and dressed with
pepper and vinegar, and then
thrown out, as good for nothing.
```

MicroEMACS automatically moved your cursor to the next line when you typed a space character

after the 30th column on your screen.

If you wish to fix the border at some special point on your screen but do not wish to go through the tedium of figuring out how many columns from the left it is, simply position the cursor where you want the border to be, type **<ctrl-X>F**, and then type a carriage return. When **<ctrl-X>F** is typed without being preceded by a **<ctrl-U>** command, it sets the word-wrap border at the point your cursor happens to be positioned. When you do this, MicroEMACS will then print a message at the bottom of your terminal that tells you where the word-wrap border is now set.

If you wish to turn off the word wrap feature again, simply set the word wrap border to one.

Search and Reverse Search

When you edit a large text, you may wish to change particular words or phrases. To do this, you can roll through the text and read each line to find them; or you can have MicroEMACS find them for you. Before you continue, close the present file by typing **<ctrl-X> <ctrl-C>**; now, reinvoke the editor to edit the file **example1.c**, as you did before. The following sections will perform some exercises with this file.

Search forward

<ctrl-S>	Search forward incrementally
<esc>S	Search forward with prompt

As you can see from the display, MicroEMACS has two ways to search forward: incrementally, and with a prompt.

An *incremental* search is one in which the search is performed as you type the characters. To see how this works, first type the **beginning of text** command *<esc><* to move the cursor to the upper left-hand corner of your screen. Now, type the **incremental search** command *<ctrl-S>*. MicroEMACS will respond by prompting with the message

```
i-search forward:
```

at the bottom of the screen.

We will now search for the pointer ***msg**. Type the letters ***msg** one at a time, starting with *. The cursor has jumped to the first place that a * was found: at the second character of the first line. The cursor moves forward in the text file and the message at the bottom of the screen changes to reflect what you have typed.

Now type **m**. The cursor has jumped ahead to the letter **s** in ***msg**. Type **s**. The cursor has jumped ahead to the letter **g** in ***msg**. Finally, type **g**. The cursor is over the space after the token ***msg**. Finally, type **<esc>** to end the string. MicroEMACS will reply with the message

```
[Done]
```

which indicates that the search is finished.

If you attempt an incremental search for a word that is not in the file, MicroEMACS will find as many of the letters as it can, and then give you an error message. For example, if you tried to search incrementally for the word ***msgs**, MicroEMACS would move the cursor to the phrase ***msg**; when you typed 's', it would tell you

```
failing i-search forward: *msgs
```

With the *prompt search*, however, you type in the word all at once. To see how this works, type **<esc><**, to return to the top of the file. Now, type the *prompt search* command **<esc>S**. MicroEMACS will respond by prompting with the message

Let's C

```
Search [*msgs]:
```

at the bottom of the screen. The word ***msgs** is shown because that was the last word for which you searched, and so it is kept in the search buffer.

Type in the words *editing text*, then press the carriage return. Notice that the cursor has jumped to the period after the word *text* in the next to last line of your text. MicroEMACS searched for the words *editing text*, found them, and moved the cursor to them.

If the word you were searching for was not in your text, or at least was not in the portion that lies between your cursor and the end of the text, MicroEMACS would not have moved the cursor, and would have displayed the message

```
Not found
```

at the bottom of your screen.

Reverse search

<ctrl-R>	Search backwards incrementally
<esc>R	Search backwards with prompt

The search commands, useful as they are, can only search forward through your text. To search backwards, use the reverse search commands **<ctrl-R>** and *<esc>R*. These work exactly the same as their forward-searching counterparts, except that they search toward the beginning of the file rather than toward the end.

For example, type *<esc>R*. MicroEMACS will reply with the message

```
Reverse search [editing text]:
```

at the bottom of your screen. The words in square brackets are the words you entered earlier for the **search** command; MicroEMACS remembered them. If you wanted to search for *editing text* again, you would just press the carriage return. For now, however, type the word *program* and press the carriage return.

Notice that the cursor has jumped so that it is under the letter *p* of the word *program* in line 1. When you search forward, the cursor will move to the **space after** the word you are searching for, whereas when you reverse search, the cursor will be moved to the **first letter** of the word you are searching for.

Cancel a command

<ctrl-G>	Cancel a search command

As you have noticed, the commands to move the cursor or to delete or kill text all execute immediately. Although this speeds your editing, it also means that if you type a command by mistake, it executes before you can stop it.

The *search* and *reverse search* commands, however, wait for you to respond to their prompts before they execute. If you type *<esc>S* or *<esc>R* by accident, MicroEMACS will interrupt your editing and wait for you to initate a search that you do not want to perform. You can evade this problem, however, with the **cancel** command *<ctrl-G>*. This command tells MicroEMACS to ignore the previous command.

To see how this command works, type *<esc>R*. When the prompt appears at the bottom of your screen, type *<ctrl-G>*. Three things happen: your terminal beeps, the characters *^G* appear at the bottom of your screen, and the cursor returns to where it was before you first typed *<esc>R*. The *<esc>R* command has been cancelled, and you are free to continue editing.

If you cancel an *incremental search* command, **<ctrl-S>** or **<esc-S>**, the cursor will return to where it was before you began the search. For example, type **<esc><** to return to the top of the file. Now type **<ctrl-S>** to begin an incremental search, and type **m**. When the cursor moves to the **m** in **simple**, type **<ctrl-G>**. The bell will ring, and your cursor will be returned to the top of the file, which is where you began the search.

Search and replace

<esc>% Search and replace

MicroEMACS also gives you a powerful function that allows you to search for a string and replace it with a keystroke. You can do this by executing the **search and replace** command **<esc>%**.

To see how this works, move to the top of the text file by typing **<esc><**; then type **<esc>%**. You will see the following message at the bottom of your screen:

```
Old string:
```

As an exercise, type **msg**. MicroEMACS will then ask:

```
New string:
```

Type **message**, and press the carriage return. As you can see, MicroEMACS jumps to the first occurrence of the string **msg**, and prints the following message at the bottom of your screen:

```
Query replace: [msg] -> [message]
```

MicroEMACS is asking if it should proceed with the replacement. Type a carriage return: this displays the options that are available to you at the bottom of your screen:

```
<SP>[,] replace, [.] rep-end, [n] dont, [!] repl rest <C-G> quit
```

The options are as follows:

Typing a space or a comma will execute the replacement, and move the cursor to the next occurrence of the old string; in this case, it will replace **msg** with **message**, and move the cursor to the next occurrence of **msg**.

Typing a period '.' will replace this one occurrence of the old string, and end the search and replace procedure; in this example, typing a period will replace this one occurrence of **msg** with **message** and end the procedure.

Typing the letter 'n' tells MicroEMACS *not* to replace this instance of the old string, and move to the next occurrence of the old string; in this case, typing 'n' will *not* replace **msg** with **message**, and the cursor will jump to the next place where **msg** occurs.

Typing an exclamation point '!' tells MicroEMACS to replace all instances of the old string with the new string, without checking with you any further. In this example, typing '!' will replace all instances of **msg** with **message** without further queries from MicroEMACS.

Finally, typing **<ctrl-G>** aborts the search and replace procedure.

Saving text and exiting

This set of basic editing commands allows you to save your text and exit from the MicroEMACS program. They are as follows:

<ctrl-X><ctrl-S>	Save text
<ctrl-X><ctrl-W>	Write text to a new file
<ctrl-Z>	Save text and exit
<ctrl-X><ctrl-C>	Exit without saving text

Let's C

You have used two of these commands already: the **save** command *<ctrl-X><ctrl-S>* and the **quit** command *<ctrl-X><ctrl-C>*, which respectively allow you to save text or to exit from MicroEMACS without saving text. (Commands that begin with *<ctrl-X>* are called **extended** commands; they are used frequently in the advanced editing to be covered in the second half of this tutorial.)

Write text to a new file

<ctrl-X> <ctrl-W> Write text to a new file

If you wish, you may copy the text you are currently editing to a text file other than the one from which you originally took the text. Do this with the **write** command *<ctrl-X><ctrl-W>*.

To test this command, type *<ctrl-X><ctrl-W>*. MicroEMACS will display the following message on the bottom of your screen:

```
Write file:
```

MicroEMACS is asking for the name of the file to which you wish to write the text. Type *sample*. MicroEMACS will reply:

```
[Wrote 23 lines]
```

The 23 lines of your text have been copied to a new file called *sample*. The status line at the bottom of your screen has changed to read as follows:

```
-- MicroEMACS -- example1.c -- File: sample --------------
```

The significance of the change in file name will be discussed in the second half of this tutorial.

Before you copy text into a new file, be sure that you have not selected a file name that is already being used. If you do, whatever is stored under that file name will be erased, and the text created with MicroEMACS will be stored in its place.

Save text and exit

Finally, the **store** command *<ctrl-Z>* will save your text **and** move you out of the MicroEMACS editor. To see how this works, watch the bottom line of your terminal carefully and type *<ctrl-Z>*. The MS-DOS MicroEMACS has saved your text, and now you can issue commands directly to MS-DOS.

Advanced editing

The second half of this tutorial introduces the advanced features of MicroEMACS.

The techniques described here will help you execute complex editing tasks with minimal trouble. You will be able to edit more than one text at a time, display more than one text on your screen at a time, enter a long or complicated phrase repeatedly with only one keystroke, and give commands to MS-DOS without having to exit from MicroEMACS.

Before beginning, however, you must prepare a new text file. Type the following command to MS-DOS:

```
me example2.c
```

If you are using the display interface of MWS, the Mark Williams shell, invoke **example2.c** in the same way that you invoked **example1.c** earlier.

In a moment, *example2.c* will appear on your screen, as follows:

```
/* Use this program to get better acquainted
 * with the MicroEMACS interactive screen editor.
 * You can use this text to learn some of the
 * more advanced editing features of MicroEMACS.
 */

#include <stdio.h>
main()
{
        FILE *fp;
        int ch;
        int filename[20];

        printf("Enter file name: ");
        gets(filename);

        if ((fp =fopen(filename,"r")) !=NULL) {
                while ((ch = fgetc(fp)) != EOF)
                        fputc(ch, stdout);
        }

        else
                printf("Cannot open %s.\n", filename);
        fclose(fp);
}
```

Arguments

Most of the commands already described in this tutorial can be used with **arguments**. An argument is a subcommand that tells MicroEMACS to execute a command a given number of times. With MicroEMACS, arguments are introduced by typing *<ctrl-U>*.

Arguments — default values

By itself, *<ctrl-U>* sets the argument at **four**. To illustrate this, first type the **next line** command *<ctrl-N>*. By itself, this command moves the cursor down one line, from being over the '/' at the beginning of line 1, to being over the *space* at the beginning of line 2.

Now, type *<ctrl-U>*. MicroEMACS replies with the message:

```
Arg: 4
```

Now type *<ctrl-N>*. The cursor jumps down **four** lines, from the beginning of line 2 to the letter *m* of the word *main* at the beginning of line 6.

Type *<ctrl-U>*. The line at the bottom of the screen again shows that the value of the argument is four. Type *<ctrl-U>* again. Now the line at the bottom of the screen reads:

```
Arg: 16
```

Type *<ctrl-U>* once more. The line at the bottom of the screen now reads:

```
Arg: 64
```

Each time you type *<ctrl-U>*, the value of the argument is **multiplied** by four. Type the **forward** command *<ctrl-F>*. The cursor has jumped ahead 64 characters, and is now over the *i* of the word *file* in the *printf* statement in line 11.

Let's C

Selecting values

Naturally, arguments do not have to be powers of four. You can set the argument to whatever number you wish, simply by typing *<ctrl-U>* and then typing in the number you want.

For example, type *<ctrl-U>*, and then type *3*. The line at the bottom of the screen now reads:

```
Arg: 3
```

Type the **delete** command *<esc>D*. MicroEMACS has deleted three words to the right.

Arguments can be used to increase the power of any **cursor movement** command, or any **kill** or **delete** command. The sole exception is *<ctrl-W>*, the **block kill** command.

Deleting with arguments—an exception

Killing and **deleting** were described in the first part of this tutorial. They were said to differ in that text that was killed was stored in a special area of the computer and could be yanked back, whereas text that was deleted was erased outright. However, there is one exception to this rule: any text that is deleted using an argument can also be yanked back.

Move the cursor to the upper left-hand corner of the screen by typing the **begin text** command *<esc><*. Then, type *<ctrl-U> 5 <ctrl-D>*. The word *Use* has disappeared. Move the cursor to the right until it is between the words *better* and *acquainted*, then type *<ctrl-Y>*. The word *Use* has been moved within the line (although the spaces around it have not been moved). This function is very handy, and should greatly speed your editing.

Remember, too, that unless you move the cursor between one set of deletions and another, the computer's storage area will not be erased, and you may yank back a jumble of text.

Buffers and files

Before beginning this section, replace the changed copy of the text on your screen with a fresh copy. Type the **quit** command *<ctrl-X><ctrl-C>* to exit from MicroEMACS without saving the text; then return to MicroEMACS to edit the file **example2.c**, as you did earlier.

Now, look at the status line at the bottom of your screen. It should appear as follows:

```
-- MicroEMACS -- example2.c -- File: example2.c --------------
```

As noted in the first half of this tutorial, the name on the left of the command line is that of the program. The name in the middle is the name of the **buffer** with which you are now working, and the name to the right is the name of the **file** from which you read the text.

Definitions

A **file** is a text that has been given a name and has been permanently stored by your computer. A **buffer** is a portion of the computer's memory that has been set aside for you to use, which may be given a name, and into which you can put text temporarily. You can put text into the buffer by typing it in from your keyboard or by **copying** it from a file.

Unlike a file, a buffer is not permanent: if your computer were to stop working (because you turned the power off, for example), a file would not be affected, but a buffer would be erased.

You must **name** your files because you work with many different files, and you must have some way to tell them apart. Likewise, MicroEMACS allows you to **name** your buffers, because MicroEMACS allows you to work with more than one buffer at a time.

File and buffer commands

MicroEMACS gives you a number of commands for handling files and buffers. These include the following:

<ctrl-X><ctrl-W>	Write text to file
<ctrl-X><ctrl-F>	Rename file
<ctrl-X><ctrl-R>	Replace buffer with named file
<ctrl-X><ctrl-V>	Switch buffer or create a new buffer
<ctrl-X>K	Delete a buffer
<ctrl-X><ctrl-B>	Display the status of each buffer

Write and rename commands

The **write** command *<ctrl-X><ctrl-W>* was introduced earlier when the commands for saving text and exiting were discussed. To review, *<ctrl-X><ctrl-W>* changes the name of the file into which the text is saved, and then writes a copy of the text into that file.

Type *<ctrl-X><ctrl-W>*. MicroEMACS responds by printing

```
Write file:
```

on the last line of your screen.

Type *junkfile*, then **<return>**. Two things happen: First, MicroEMACS writes the message

```
[Wrote 21 lines]
```

at the bottom of your screen. Second, the name of the file shown on the status line has changed from *example2.c* to *junkfile*. MicroEMACS is reminding you that your text is now being saved into the file **junkfile**.

The **file rename** command *<ctrl-X><ctrl-F>* allows you rename the file to which you are saving text, **without** automatically writing the text to it. Type *<ctrl-X><ctrl-F>*. MicroEMACS will reply with the prompt:

```
Name:
```

Type **example2.c** and **<return>**. MicroEMACS does **not** send you a message that lines were written to the file; however, the name of the file shown on the status line has changed from *junkfile* back to *example2.c*.

Replace text in a buffer

The **replace** command *<ctrl-X><ctrl-R>* allows you to replace the text in your buffer with the text taken from another file.

Suppose, for example, that you had edited *example2.c* and saved it, and now wished to edit *example1.c*. You could exit from MicroEMACS, then re-invoke MicroEMACS for the file *example2.c*, but this is cumbersome. A more efficient way is to simply replace the *example2.c* in your buffer with *example1.c*.

Type *<ctrl-X><ctrl-R>*. MicroEMACS replies with the prompt:

```
Read file:
```

Type *example1.c*. Notice that *example2.c* has rolled away and been replaced with *example1.c*. Now, check the status line. Notice that although the name of the **buffer** is still *example2.c*, the name of the **file** has changed to *example1.c*. You can now edit *example1.c*; when you save the edited text, MicroEMACS will copy it back into the file *example1.c* — unless, of course, you again choose to

rename the file.

Visiting another buffer

The last command of this set, the **visit** command <*ctrl-X*><*ctrl-V*>, allows you to create more than one buffer at a time, to jump from one buffer to another, and move text between buffers. This powerful command has numerous features.

Before beginning, however, straighten up your buffer by replacing *example1.c* with *example2.c*. Type the **replace** command <*ctrl-X*><*ctrl-R*>; when MicroEMACS replies by asking

```
Read file:
```

at the bottom of your screen, type *example2.c*.

You should now have the file *example2.c* read into the buffer named *example2.c*.

Now, type the **visit** command <*ctrl-X*><*ctrl-V*>. MicroEMACS replies with the prompt

```
Visit file:
```

at the bottom of the screen. Now type *example1.c*. Several things happen. *example2.c* rolls off the screen and is replaced with *example1.c*; the status line changes to show that both the buffer name and the file name are now *example1.c*; and the message

```
[Read 23 lines]
```

appears at the bottom of the screen.

This does **not** mean that your previous buffer has been erased, as it would have been had you used the **replace** command <*ctrl-X*><*ctrl-R*>. *example2.c* is still being kept "alive" in a buffer and is available for editing; however, it is not being shown on your screen at the present moment.

Type <*ctrl-X*><*ctrl-V*> again, and when the prompt appears, type *example2.c*. *example1.c* scrolls off your screen and is replaced by *example2.c*, and the message

```
[Old buffer]
```

appears at the bottom of your screen. You have just jumped from one buffer to another.

Move text from one buffer to another

The **visit** command <*ctrl-X*><*ctrl-V*> not only allows you to jump from one buffer to another, it allows you to **move text** from one buffer to another as well. The following example shows how you can do this.

First, kill the first line of *example2.c* by typing the **kill** command <*ctrl-K*> twice. This removes both the line of text **and** the space that it occupied; if you did not remove the space as well the line itself, no new line would be created for the text when you yank it back. Next, type <*ctrl-X*><*ctrl-V*>. When the prompt

```
Visit file:
```

appears at the bottom of your screen, type *example1.c*. When *example1.c* has rolled onto your screen, type the **yank back** command <*ctrl-Y*>. The line you killed in *example2.c* has now been moved into *example1.c*.

Checking buffer status

The number of buffers you can use at any one time is limited only by the size of your computer. You should create only as many buffers as you need to use immediately; this will help the computer run efficiently.

To help you keep track of your buffers, MicroEMACS has the **buffer status** command *<ctrl-X><ctrl-B>*. Type *<ctrl-X><ctrl-B>*. The status line has moved up to the middle of the screen, and the bottom half of your screen has been replaced with the following display:

```
C    Size Lines   Buffer         File
-    ---- -----   ------         ----
*     655    24   example1.c     example1.c
*     403    20   example2.c     example2.c
```

This display is called the **buffer status window**. The use of windows will be discussed more fully in the following section.

The letter *C* over the leftmost column stands for *Changed*. An asterisk on a line indicates that the buffer has been changed since it was last saved, whereas a space means that the buffer has not been changed. *Size* indicates the buffer's size, in number of characters; *Buffer* lists the buffer name, and *File* lists the file name.

Now, kill the second line of *example1.c* by typing the **kill** command *<ctrl-K>*. Then type *<ctrl-X><ctrl-B>* once again. The size of the buffer *example1.c* has been reduced from 657 characters to 595 to reflect the decrease in the size of the buffer.

To make this display disappear, type the **one window** command *<ctrl-X>1*. This command will be discussed in full in the next section.

Renaming a buffer

One more point must be covered with the **visit** command. MS-DOS will not allow you to have more than one file with the same name. For the same reason, MicroEMACS will not allow you to have more than one **buffer** with the same name.

Ordinarily, when you visit a file that is not already in a buffer, MicroEMACS will create a new buffer and give it the same name as the file you are visiting. However, if for some reason you already have a buffer with the same name as the file you wish to visit, MicroEMACS will stop and ask you to give a new, different name to the buffer it is creating.

For example, suppose that you wanted to visit a new *file* named **sample**, but you already had a **buffer** named *sample*. MicroEMACS would stop and give you this prompt at the bottom of the screen:

```
Buffer name:
```

You would type in a name for this new buffer. This name could not duplicate the name of any existing buffer. MicroEMACS would then read the file **sample** into the newly named buffer.

Delete a buffer

If you wish to delete a buffer, simply type the **delete buffer** command *<ctrl-X>K*. This command will allow you to delete only a buffer that is hidden, not one that is being displayed.

Type *<ctrl-X>K*. MicroEMACS will give you the prompt:

```
Kill buffer:
```

Type *example2.c*. Because you have changed the buffer, MicroEMACS asks:

```
Discard changes [y/n]?
```

Type *y*. Then type the **buffer status** command *<ctrl-X><ctrl-B>*; the buffer status window will no longer show the buffer *example2.c*. Although the prompt refers to **killing** a buffer, the buffer is in fact **deleted** and cannot be yanked back.

Windows

Let's C

Before beginning this section, it will be necessary to create a new text file. Exit from MicroEMACS by typing the **quit** command *<ctrl-X><ctrl-C>*; then reinvoke MicroEMACS for the text file *example1.c* as you did earlier.

Now, copy *example2.c* into a buffer by typing the *visit* command *<ctrl-X><ctrl-V>*. When the message

```
Visit file:
```

appears at the bottom of your screen, type *example2.c*. MicroEMACS will read *example2.c* into a buffer, and show the message

```
[Read 21 lines]
```

at the bottom of your screen.

Finally, copy a new text, called *example3.c*, into a buffer. Type *<ctrl-X><ctrl-V>* again. When MicroEMACS asks which file to visit, type *example3.c*. The message

```
[Read 123 lines]
```

will appear at the bottom of your screen.

The first screenful of text will appear as follows:

```
/*
 * Factor prints out the prime factorization of numbers.
 * If there are any arguments, then it factors these.  If
 * there are no arguments, then it reads stdin until
 * either EOF or the number zero or a non-numeric
 * non-white-space character.  Since factor does all of
 * its calculations in double format, the largest number
 * which can be handled is quite large.
 */
#include <stdio.h>
#include <math.h>
#include <ctype.h>

#define NUL '\0'
#define ERROR 0x10 /* largest input base */
#define MAXNUM 200 /* max number of chars in number */

main(argc, argv)
int argc;
register char *argv[];

-- MicroEMACS -- example3.c -- File: example3.c --------------
```

At this point, *example3.c* is on your screen, and *example1.c* and *example2.c* are hidden.

You could edit first one text and then another, while remembering just how things stood with the texts that were hidden; but it would be much easier if you could display all three texts on your screen simultaneously. MicroEMACS allows you to do just that by using **windows**.

Creating windows and moving between them

A **window** is a portion of your screen that is set aside and can be manipulated independently from the rest of the screen. The following commands let you create windows and move between them:

<ctrl-X>2	Create a window
<ctrl-X>1	Delete extra windows
<ctrl-X>N	Move to next window
<ctrl-X>P	Move to previous window

The best way to grasp how a window works is to create one and work with it. To begin, type the **create a window** command *<ctrl-X>2*.

Your screen is now divided into two parts, an upper and a lower. The same text is in each part, and the command lines give *example3.c* for the buffer and file names. Also, note that you still have only one cursor, which is in the upper left-hand corner of the screen.

The next step is to move from one window to another. Type the **next window** command *<ctrl-X>N*. Your cursor has now jumped to the upper left-hand corner of the **lower** window.

Type the **previous window** command *<ctrl-X>P*. Your cursor has returned to the upper left-hand corner of the top window.

Now, type *<ctrl-X>2* again. The window on the top of your screen is now divided into two windows, for a total of three on your screen. Type *<ctrl-X>2* again. The window at the top of your screen has again divided into two windows, for a total of four.

It is possible to have as many as 11 windows on your screen at one time, although each window will show only the control line and one or two lines of text. Neither *<ctrl-X>2* nor *<ctrl-X>1* can be used with arguments.

Now, type the **one window** command *<ctrl-X>1*. All of the extra windows have been eliminated, or **closed**.

Enlarging and shrinking windows

When MicroEMACS creates a window, it divides the window in which the cursor is positioned into half. You do not have to leave the windows at the size MicroEMACS creates them, however. If you wish, you may adjust the relative size of each window on your screen, using the **enlarge window** and **shrink window** commands:

<ctrl-X>Z	Enlarge window
<ctrl-X><ctrl-Z>	Shrink window

To see how these work, first type *<ctrl-X>2* twice. Your screen is now divided into three windows: two in the top half of your screen, and the third in the bottom half.

Now, type the **enlarge window** command *<ctrl-X>Z*. The window at the top of your screen is now one line bigger: it has borrowed a line from the window below it. Type *<ctrl-X>Z* again. Once again, the top window has borrowed a line from the middle window.

Now, type the **next window** command *<ctrl-X>N* to move your cursor into the middle window. Again, type the **enlarge window** command *<ctrl-X>Z*. The middle window has borrowed a line from the bottom window, and is now one line larger.

The **enlarge window** command *<ctrl-X>Z* allows you to enlarge the window your cursor is in by borrowing lines from another window, provided that you do not shrink that other window out of existence. Every window must have at least two lines in it: one command line and one line of text.

The **shrink window** command *<ctrl-X><ctrl-Z>* allows you to decrease the size of a window. Type *<ctrl-X><ctrl-Z>*. The present window is now one line smaller, and the lower window is one line larger because the line borrowed earlier has been returned.

Let's C

The **enlarge window** and **shrink window** commands can also be used with arguments introduced with *<ctrl-U>*. However, remember that MicroEMACS will not accept an argument that would shrink another window out of existence.

Displaying text within a window

Displaying text within the limited area of a window can present special problems. The **view** commands *<ctrl-V>* and *<esc>V* will roll window-sized portions of text up or down, but you may become disoriented when a window shows only four or five lines of text at a time. Therefore, three special commands are available for displaying text within a window:

<ctrl-X><ctrl-N>	Scroll down
<ctrl-X><ctrl-P>	Scroll up
<esc>!	Move within window

Two commands allow you to move your text by one line at a time, or **scroll** it: the **scroll up** command *<ctrl-X><ctrl-N>*, and the **scroll down** command *<ctrl-X><ctrl-P>*.

Type *<ctrl-X><ctrl-N>*. The line at the top of your window has vanished, a new line has appeared at the bottom of your window, and the cursor is now at the beginning of what had been the second line of your window.

Now type *<ctrl-X><ctrl-P>*. The line at the top that had vanished earlier has now returned, the cursor is at the beginning of it, and the line at the bottom of the window has vanished. These commands allow you to move forward in your text slowly so that you do not become disoriented.

Both of these commands can be used with arguments introduced by *<ctrl-U>*.

The third special movement command is the **move within window** command *<esc>!*. This command moves the line your cursor is on to the top of the window.

To try this out, move the cursor down three lines by typing *<ctrl-U>3<ctrl-N>*, then type *<esc>!*. (Be sure to type an exclamation point '!', not a numeral one '1', or nothing will happen.) The line to which you had moved the cursor is now the first line in the window, and three new lines have scrolled up from the bottom of the window. You will find this command to be very useful as you become more experienced at using windows.

All three special movement commands can also be used when your screen has no extra windows, although you will not need them as much.

One buffer

Now that you have been introduced to the commands for manipulating windows, you can begin to use windows to speed your editing.

To begin with, scroll up the window you are in until you reach the top line of your text. You can do this either by typing the **scroll up** command *<ctrl-X><ctrl-P>* several times, or by typing *<esc><*.

Kill the first line of text with the **kill** command *<ctrl-K>*. The first line of text has vanished from all three windows. Now, type *<ctrl-Y>* to yank back the text you just killed. The line has reappeared in all three windows.

The main advantage to displaying one buffer with more than one window is that each window can display a different portion of the text. This can be quite helpful if you are editing or moving a large text.

To demonstrate this, do the following: First, move to the end of the text in your present window by typing the *end of text* command **<esc>>**, then typing the *previous line* command **<ctrl-P>** four times. Now, kill the last four lines.

You could move the killed lines to the beginning of your text by typing the **beginning of text** command *<esc><*; however, it is more convenient simply to type the **next window** command *<ctrl-X>N*, which will move you to the beginning of the text as displayed in the next window. MicroEMACS remembers a different cursor position for each window.

Now yank back the four killed lines by typing *<ctrl-Y>*. You can simultaneously observe that the lines have been removed from the end of your text and that they have been restored at the beginning.

Multiple buffers

Windows are especially helpful when they display more than one text. Remember that at present you are working with **three** buffers, named *example1.c*, *example2.c*, and *example3.c*, although your screen is displaying only *example3.c*. To display a different text in a window, use the **switch buffer** command *<ctrl-X>B*.

Type *<ctrl-X>B*. When MicroEMACS asks

```
Use buffer:
```

at the bottom of the screen, type *example1.c*. The text in your present window will be replaced with *example1.c*. The command line in that window has changed, too, to reflect the fact that the buffer and the file names are now *example1.c*.

Moving and copying text among buffers

It is now very easy to copy text among buffers. To see how this is done, first kill the first line of *example1.c* by typing the *<ctrl-K>* command twice. Yank back the line immediately by typing *<ctrl-Y>*. Remember, the line you killed has **not** been erased from its special storage area, and may be yanked back any number of times.

Now, move to the previous window by typing *<ctrl-X>P*, then yank back the killed line by typing *<ctrl-Y>*. This technique can also be used with the *block kill* command **<ctrl-W>** to move large amounts of text from one buffer to another.

Checking buffer status

The **buffer status command** *<ctrl-X><ctrl-B>* can be used when you are already displaying more than one window on your screen.

When you want to remove the buffer status window, use either the **one window** command *<ctrl-X>1*, or move your cursor into the buffer status window using the **next window** command *<ctrl-X>N* and replace it with another buffer by typing the **switch buffer** command *<ctrl-X>B*.

Saving text from windows

The final step is to save the text from your windows and buffers. Close the lower two windows with the **one window** command *<ctrl-X>1*. Remember, when you close a window, the text that it displayed is still kept in a buffer that is **hidden** from your screen. For now, do *not* save any of these altered texts.

When you use the **save** command *<ctrl-X><ctrl-S>*, only the text in the window in which the cursor is positioned will be written to its file. If only one window is displayed on the screen, the **save** command will save only its text.

If you made changes to the text in another buffer, such as moving portions of it to another buffer, MicroEMACS will ask

```
Quit [y/n]:
```

If you answer '**n**', MicroEMACS will *save* the contents of the buffer you are currently displaying by writing them to your disk, but it will ignore the contents of other buffers, and your cursor will be

returned to its previous position in the text. If you answer '**y**', MicroEMACS again will save the contents of the current buffer and ignore the other buffers, but you will exit from MicroEMACS and return to MS-DOS. Exit from MicroEMACS by typing the **quit** command *<ctrl-X><ctrl-C>*.

Keyboard macros

Another helpful feature of MicroEMACS is that it allows you to create a **keyboard macro**.

Before beginning this section, reinvoke MicroEMACS to edit *example3.c* as you did earlier.

The term **macro** means a number of commands or characters that are bundled together under a common name. Although MicroEMACS allows you to create only one macro at a time, this macro can consist of a common **phrase** or a common **command** or **series of commands** that you use while editing your file.

Keyboard macro commands

The keyboard macro commands are as follows:

<ctrl-X>(	Begin macro collection
<ctrl-X>)	End macro collection
<ctrl-X>E	Execute macro

To begin to create a macro, type the **begin macro** command *<ctrl-X>(*. Be sure to type an open parenthesis '(', not a numeral '9'. MicroEMACS will reply with the message

```
[Start macro]
```

Type the following phrase:

```
MAXNUM
```

Then type the **end macro** command *<ctrl-X>)*. Be sure you type a close parenthesis ')', not a numeral '0'. MicroEMACS will reply with the message

```
[End macro]
```

Move your cursor down two lines and execute the macro by typing the **execute macro** command *<ctrl-X>E*. The phrase you typed into the macro has been inserted into your text.

Should you give these commands in the wrong order, MicroEMACS will warn you that you are making a mistake. For example, if you open a keyboard macro by typing **<ctrl-X>(**, and then attempt to open another keyboard macro by again typing **<ctrl-X>(**, MicroEMACS will say:

```
Not now
```

Should you accidentally open a keyboard macro, or enter the wrong commands into it, you can cancel the entire macro simply by typing **<ctrl-G>**.

Replacing a macro

To replace this macro with another, go through the same process. Type *<ctrl-X>(*. Then type the **buffer status** command *<ctrl-X><ctrl-B>*, and type *<ctrl-X>)*. Remove the buffer status window by typing the **one window** command *<ctrl-X>1*.

Now execute your keyboard macro by typing the *execute macro* command *<ctrl-X>E*. The **buffer status** command has executed once more.

Whenever you exit from MicroEMACS, your keyboard macro is erased, and must be retyped when you return.

Sending commands to MS-DOS

The only remaining command you need to learn is the **program interrupt** command *<ctrl-X>!*. This command allows you to interrupt your editing, give a command directly to MS-DOS, and then resume editing without affecting your text in any way.

The command *<ctrl-X>!* allows you to send **one** command to the operating system. To see how this command works, type *<ctrl>!*. The prompt *MS-DOS command:* has appeared at the bottom of your screen. Type *dir*. Observe that the directory's table of contents scrolls across your screen. To return to your editing, simply type a carriage return.

Compiling and debugging through MicroEMACS

MicroEMACS can be used with the compilation command **cc** to give you a reliable system for debugging new programs.

Often, when you're writing a new program, you face the situation in which you try to compile, but the compiler produces error messages and aborts the compilation. You must then invoke your editor, change the program, close the editor, and try the compilation over again. This cycle of compilation—editing—recompilation can be quite bothersome.

To remove some of the drudgery from compiling, the **cc** command has the *automatic*, or MicroEMACS option, **-A**. When you compile with this option, the MicroEMACS screen editor will be invoked automatically if any errors occur. The error or errors generated during compilation will be displayed in one window, and your text in the other, with the cursor set at the number of the line that the compiler indicated had the error.

Try the following example. Use MicroEMACS to enter the following program, which you should call **error.c**:

```
main() {
        printf("Hello, world!\n")
}
```

The semicolon was left off of the **printf** statement, which is an error. Now, try compiling **error.c** with the following **cc** command:

```
cc -A error.c
```

You should see no messages from the compiler because they are all being diverted into a buffer to be used by MicroEMACS. Then MicroEMACS will appear automatically. In one window you should see the message:

```
3: missing ';'
```

and in the other you should see your source code for **error.c**, with the cursor set on line 3.

If you had more than one error, typing **<ctrl-X>>** would move you to the next line with an error in it; typing **<ctrl-X><** would return you to the previous error. With some errors, such as those for missing braces or semicolons, the compiler cannot always tell exactly which line the error occurred on, but it will almost always point to a line that is near the source of the error.

Now, correct the error by typing a semicolon at the end of line 2. Close the file by typing **<ctrl-Z>**. **cc** will be invoked again automatically.

cc will continue to compile your program either until the program compiles without error, or until you exit from MicroEMACS by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

The MicroEMACS help facility

MicroEMACS has a built-in help function. With it, you can ask for information either for a word that you type in, or for a word over which the cursor is positioned. The MicroEMACS help file contains the bindings for all library functions and macros included with **Let's C**.

For example, consider that you are preparing a C program and want more information about the function **fopen**. Type **<ctrl-X>?**. At the bottom of the screen will appear the prompt

```
Topic:
```

Type **fopen**. MicroEMACS will search its help file, find its entry for **fopen**, then open a window and print the following:

```
fopen - Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (name, type) char *name, *type;
```

If you wish, you can kill the information in the help window and copy it into your program to ensure that you prepare the function call correctly.

Consider, however, that you are checking a program written earlier, and you wish to check the call to **fopen**. Simply move the cursor until it is positioned over one of the letters in **fopen**, then type **<esc>?**. MicroEMACS will open its help window, and show the same information it did above.

To erase the help window, type **<esc>2**.

Where to go from here

For a complete summary of MicroEMACS's commands, see the entry for **me** in the Lexicon.

The next section introduces **make**, a utility is helpful in building and maintaining large programs.

make Programming Discipline

make is a utility that relieves you of the drudgery of building a complex C program.

How does make work?

To understand how **make** works, it is first necessary to understand how a C program is built: how **Let's C** takes you from the C source code that you write to the executable program that you can run on your computer.

The file of C source code that you write is called a *source module*. When **Let's C** compiles a source module, it uses the C code in the source module, plus the code in the header files that the code calls to produce an *object module*. This object module is *not* executable by itself. To create an *executable file*, the object module generated from your source module must be handed to a linker, which links the code in the object module with the appropriate library routines that the object module calls, and adds the appropriate C runtime startup routine.

For example, consider the following C program, called **hello.c**:

```
main()
{
        printf("Hello, world\n");
}
```

When **Let's C** compiles the file that contains C code shown above, it generates an object module called **hello.obj**. This object module is not executable because it does not contain the code to execute the function **printf**; that code is contained in a library. To create an executable program, you must hand **hello.obj** to the linker **ld**, which copies the code for **printf** from a library and into your program, adds the appropriate C runtime startup routine, and writes the executable file called **hello.exe**. This third file, **hello.exe**, is what you can execute on your computer.

The term *dependency* describes the relationship of executable file to object module to source module. The executable program *depends* on the object module, the library, and the C runtime startup. The object module, in turn, depends on the source module and its header files (if any).

A program like **hello.exe** has a simple set of dependencies: the executable file is built from one object module, which in turn is compiled from one source module. If you changed the source module **hello.c**, creating an updated version of **hello.exe** would be easy: you would simply compile **hello.c** to create **hello.obj**, which you would link with the library and the runtime startup to create **hello.exe**. **Let's C**, in fact, does this for you automatically: all you need to do is type

```
cc hello.c
```

and **Let's C** takes care of everything.

On the other hand, the dependencies of a large program can be very complex. For example, the executable file for the MicroEMACS screen editor is built from several dozen object modules, each of which is compiled from a source module plus one or more header files. Updating a program as large as MicroEMACS, even when you change only one source module, can be quite difficult. To rebuild its executable file by hand, you must remember the names of all of the source modules used, compile them, and link them into the executable file. Needless to say, it is very inefficient to recompile several dozen object modules to create an executable when you have changed only one of them.

make automatically rebuilds large programs for you. You prepare a file, called a **makefile**, that describes your program's chain of dependencies. **make** then reads your **makefile**, checks to see which source modules have been updated, recompiles only the ones that have been changed, and then relinks all of the object modules to create a new executable file. **make** both saves you time, because it recompiles only the source modules that have changed, and spares you the drudgery of rebuilding your large program by hand.

Try make

The following example shows how easy it is to use **make**.

To begin, **make** examines the time and date that MS-DOS has stamped on each source file and object module. When you edit a source module, MS-DOS marks it with the time at which you edited it. Thus, if a source module has a time that is *later* than that of its corresponding object module, then **make** knows that the source module was changed since the object module was last compiled and it will compile a new object module from the altered source module. If you do not reset the time on your system whenever you reboot, *every time,* some files will not have the correct date and time and **make** cannot work correctly.

To see how **make** works, try compiling a program called **factor**. It is built from the following files:

```
atod.c
factor.c
makefile
```

All three are included with your copy of **Let's C**.

If you do not have a hard disk, insert disk 8 (which holds the sample programs) into drive B, and make sure that disk 2 (the compiler disk) is in drive A. Use the **cd** command to shift into directory **src**.

Now, type **make**. **make** will begin by reading **makefile**, which describes all of **factor**'s dependencies. It will then use the **makefile** description to create **factor**. The following will appear on your screen:

```
cc -c factor.c
cc -c atod.c
cc -f -o factor.exe factor.obj atod.obj -lm
```

Each of these messages describes an action that **make** has performed. The first shows that **make** is compiling **factor.c**, the second shows that it is compiling **atod.c**, and the third shows that it is linking the compiled object modules **atod.obj** and **factor.obj** to create the executable file **factor.exe**.

When **make** has finished, the MS-DOS prompt will return. To see how your newly compiled program works, type

```
factor 100
```

factor will calculate the prime factors of its argument **100**, and print them on the screen.

To see what happens if you try to re-make your file, type **make** again. **make** will run quietly for a moment, and then exit. **make** checked the dates and times of the object modules and their corresponding source modules and saw that the object modules had a time later than that of the source modules. Because no source module changed, there was no need to recompile an object module or relink the executable file, so **make** quietly exited.

To see what happens when one of the source modules changes, try the following. Use the MicroEMACS screen editor to open the file **factor.c** for editing. Insert the following line into the comments at the top, immediately following the **/***:

```
* This comment is for test purposes only.
```

Now exit. Type **make** once again. This time, you will see the following on your screen:

```
cc -c factor.c
cc -f -o factor.exe factor.obj atod.obj -lm
```

Because you altered the source module **factor.c**, its time was later than that of its corresponding object module, **factor.obj**. When **make** compared the times of **factor.c** and **factor.obj**, it noted that **factor.c** had been altered. It then recompiled **factor.c** and relinked **factor.obj** and **atod.obj** to re-create the executable file **factor.exe**. **make** did not touch the source module **atod.c** because **atod.c** had not been changed since the last time it was compiled.

As you can see, **make** greatly simplifies the construction of a C program that uses more than one source module.

Essential make

Although **make** is a powerful program, its basic features are easy to master. This section will show you how to construct elementary **make** scripts.

The makefile

When you invoke **make**, it searches the directories named in the environmental variable **PATH** for a file called **makefile**. As noted earlier, the **makefile** is a text file that describes a C program's dependencies. It also describes the type of program you wish to build, and the commands for building it.

A **makefile** has three basic parts.

First, the **makefile** describes the executable file's dependencies. That is, it lists the object modules needed to create the executable file. The name of the executable file is always followed by a colon ':' and then by the names of files from which the target file is generated.

For example, if the program **feud.exe** is built from the object modules **hatfield.obj** and **mccoy.obj**, you would type:

```
feud.exe:    hatfield.obj mccoy.obj
```

If the files **hatfield.obj** and **mccoy.obj** do not exist, **make** knows to create them from the source modules **hatfield.c** and **mccoy.c**.

Second, the **makefile** holds one or more *command* lines. The command line gives the command to compile the program in question. The only difference between a **makefile** command line and an ordinary **cc** command is that a **makefile** command line *must* begin with a space or a tab character.

For example, the **makefile** to generate the program **feud.exe** must contain the following command line:

```
        cc -o feud.exe hatfield.obj mccoy.obj
```

For a detailed description of the **cc** command and its options, refer to the entry for **cc** in the Lexicon.

Third, the **makefile** lists all of the header files that your program uses. These are given so that **make** can check if they were modified since your program was last compiled. For example, if the program **hatfield.c** used the header file **shotgun.h** and **mccoy.c** used the header files **rifle.h** and **pistol.h**, the **makefile** to generate **feud.exe** would include the following lines:

```
hatfield.obj: shotgun.h
mccoy.obj: rifle.h pistol.h
```

Thus, the entire **makefile** to generate the program **feud.exe** is as follows:

```
feud.exe: hatfield.obj mccoy.obj
        cc -o feud.exe hatfield.obj mccoy.obj

hatfield.obj: shotgun.h
mccoy.obj: rifle.h pistol.h
```

A **makefile** may also contain *macro definitions* and *comments*. These are described below.

Building a simple makefile

The program **factor.exe** is built from two source modules, **factor.c** and **atod.c**. No header files are used. The **makefile** contains the following two lines:

```
factor.exe: factor.obj atod.obj
        cc -f -o factor.exe factor.obj atod.obj -lm
```

The first line describes the dependency for the executable file **factor.exe** by naming the two object modules needed to build it. The second line gives the command needed to build **factor.exe**. The option **-lm** at the end of the command line tells **cc** that this program needs the mathematics library **libm** when the program is linked. No header file dependencies are described because these programs use no header files.

Comments and macros

You can embed comments within a **makefile**. A *comment* is a line of text that is ignored; this lets you "document" the file, so that whoever reads it will now know what it is for. **make** ignores all lines that begin with a pound sign '#'. For example, you may wish to include the following information in your **makefile** for **factor**:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.

        factor: factor.obj atod.obj
                cc -f -o factor.exe factor.obj atod.obj -lm
```

Anyone who reads this file will know immediately what it is for by looking at the comments.

make also lets you define macros within your **makefile**. A *macro* is a symbol that represents a string of text. Usually, a macro is defined at the beginning of the **makefile** using a *macro definition statement*. This statement uses the following syntax:

```
SYMBOL = string of text
```

Thereafter, when you use the symbol in your **makefile**, it must begin with a dollar sign '$' and be enclosed within parentheses.

Macros eliminate the chore of retyping long strings of file names. For example, with the **makefile** for the program **factor**, you may wish to use a macro to substitute for the names of the object modules out of which it is built. This is done as follows:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.
```

```
OBJ = factor.obj atod.obj
factor: $(OBJ)
        cc -o factor.exe $(OBJ) -lm
```

The macro **OBJ** is used in this **makefile**. If you use a macro that has not been defined, **make** substitutes an empty string for it. The use of a macro makes sense when generating large files out of a dozen or more source modules. You avoid retyping the source module names, and potential errors are avoided.

Setting the time

As noted above, **make** checks to see which source modules have been modified before it regenerates your C program. This is done to avoid wasteful recompiling of source modules that have not been updated.

make determines that a source module has been altered by comparing its date against that of the target program. For example, if the object module **factor.obj** was generated on March 16, 1987, 10:52:47 A.M., and the source module **factor.c** was modified on March 20, 1987, at 11:19:06 A.M., **make** will know that **factor.c** needs to be recompiled because it is *younger* than **factor.obj**.

For this reason, if you wish to use **make**, you *must* reset the date and time every time you reboot your system. Some users do not do this routinely; however, unless the time is reset *every* time, **make** will not work correctly.

Building a large program

As shown earlier, **make** can ease the task of generating a large program. The following is the **makefile** used to generate the screen editor MicroEMACS:

```
# MS-DOS limits command line tails to no more
# than 128 characters.  To skirt this limit, the
# command line is built into a temporary file,
# which we pass to make.

O1 = ansi.obj basic.obj buffer.obj display.obj file.obj \
        fileio.obj line.obj main.obj
O2 = window.obj word.obj tcap.obj
O3 = random.obj region.obj search.obj spawn.obj termio.obj vt52.obj

me.exe: $(O1) $(O2) $(O3)
        echo $(O1) > maketemp
        echo $(O2) >> maketemp
        echo $(O3) >> maketemp
        cc -o me.exe @maketemp
        del maketemp

$(O1) $(O2) $(O3): ed.h
```

This file shows how the elements of a **makefile** are used to control the generation of a large program.

The first four lines consist of comments that describe a peculiarity of the file, as fair warning to future programmers.

The next four lines define the macros **O1**, **O2**, and **O3**, which substitute for the 17 files that make up this program. Three macros must be used because, as explained in the comments, under MS-DOS no command line can have a tail longer than 128 characters.

The next line gives the name of the target file, **me.exe**, and the files needed to generate it; in this case, these file names are represented by the macros **O1**, **O2**, and **O3**.

The next three lines begin with the command **echo**. These command lines copy the three macros into the temporary file **maketemp**; this strategy is one way around the 128-character limit on command lines.

The next line is the command line. It controls the compiling of the files listed in **maketemp**.

The next to last line deletes **maketemp**, so that this file is no longer cluttering up your directory. Finally, the last line notes that all 17 of the MicroEMACS object modules are built in from the header file **ed.h**.

Command line options

Although **make** is controlled by your **makefile**, you can also control **make** by using command line options. These allow you to alter **make**'s activity without having to edit your **makefile**.

Options must follow the command name on the command line and begin with a hyphen, '-', using the following format. The square brackets merely indicate that you can select any of these options; do *not* type the brackets when you use the **make** command:

```
make [ -dinprst ] [ -f filename ]
```

Each option is described below.

-d (debug) **make** describes all of its decisions. You can use this to debug your **makefile**.

-f *filename*
(file) option tells **make** that its commands are in a file other than **makefile**. For example, the command

```
make -f smith
```

tells **make** to use the file **smith** rather than **makefile**. If you do not use this option, **make** searches the directories named in the environmental variable **PATH**, and then the current directory for a file entitled **makefile** to execute.

-i (ignore errors) **make** ignores error returns from commands and continues processing. Normally, **make** exits if a command returns an error status.

-n (no execution) **make** tests dependencies and modification times but does not execute commands. This option is especially helpful when constructing or debugging a **makefile**.

-p (print) **make** prints all macro definitions and target descriptions.

-r (rules) **make** does not use the default macros and commands from **$LIBPATH\mmacros** and **$LIBPATH\mactions**. These files will be described below.

-s (silent) **make** does not print each command line as it is executed.

-t (touch) **make** changes the modification time of each executable file and object module to the current time. This suppresses recreation of the executable file, and recompilation of the object modules. Although this option is used typically after a purely cosmetic change to a source module or after adding a definition to a header file, it must be used with great caution.

Other command line features

In addition to the options listed above, you may include other information on your command line.

First, you can define macros on the command line. A macro definition must *follow* any command line options. For example, the command line

```
make -n -f smith "CSD=-VCSD"
```

tells **make** to run in the *no execution* mode, reading the file **smith** instead of **makefile**, and defining

the macro **CSD** to mean **-VCSD**.

The ability to define macros on the command line means that you can create a **makefile** using macros that are not yet defined; this greatly increases **make**'s flexibility and makes it even more helpful in creating and debugging large programs. In the above example, you can define a command line as follows:

```
cc $(CSD) example.c
```

When you define the macro **CSD** on the command line, then the program is compiled using the **-VCSD** option, which creates an executable that can be debugged with **csd**, the Mark Williams C Source Debugger. If the macro is not set, however, then it is simply skipped when the command line is executed, and the program is compiled in the usual manner.

Another command-line feature is the ability to change the name of the *target file* on the command line. Normally, the target file is the executable file that you wish to create, although, as will be seen, it does not have to be. As will be discussed below, a **makefile** can name more than one target file. **make** normally assumes that the target is the first target file named in **makefile**. However, the command line may name one or more target files at the end of the line, after any options and any macro definitions.

To see how this works, recall the program **factor** described above. **factor** is generated out of the source modules **factor.c** and **atod.c**. The command

```
make atod.obj
```

with the **makefile** outlined above would produce the following **cc** command line:

```
cc -c atod.c
```

if the object module **atod.obj** does not exist or is outdated. Here, **make** compiles **atod.c** to create the target specified in the **make** command line, that is, **atod.obj**, but it does not create **factor**. This feature allows you to apply your **makefile** to only a portion of your program.

The use of special, or *alternative*, target files is discussed below.

Advanced make

This section describes some of **make**'s advanced features. For most of your work, you will not need these features; however, if you create an extremely complex program, you will find them most helpful.

Default rules

The operation of **make** is governed by a set of *default rules*. These rules were designed to simplify the compilation of a typical program; however, unusual tasks may require that you bypass or alter the default rules.

To begin, **make** uses information from the files **mmacros** and **mactions** to define default macros and compilation commands. **make** looks for these files in the directories named in the environmental variable **LIBPATH**. **make** uses the commands in **mmacros** and **mactions** whenever the **makefile** specifies no explicit regeneration commands. The command line option **-r** tells **make** not to use the macros and actions defined in **mmacros** and **mactions**.

As shown in earlier examples, **make** knows by default to generate the object module **atod.obj** from the source module **atod.c** with the command

```
cc -c atod.c
```

The macro **.SUFFIXES** defines the suffixes **make** knows about by default. Its definition in **mmacros** includes both the **.obj** and **.c** suffixes.

make's files **mmacros** and **mactions** use pre-defined macros to increase their scope and flexibility. These are as follows:

$< This stands for the name of the file or files that cause the action of a default rule. For example, if you altered the file **atod.c** and then invoked **make** to rebuild the executable file **factor.exe**, **$<** would then stand for **atod.c**.

$*** This stands for the name of the target of a default rule with its suffix removed. If it had been used in the above example, **$* would have stood for **atod**.

$<** and **$* work *only* with default rules; these macros will not work in a **makefile**.

$? This stands for the names of the files that cause the action and that are younger than the target file.

$@ This stands for the target name.

You can use the macros **$?** and **$@** in a **makefile**. For example, the following rule updates the file **factor** with the objects defined by macro **$(OBJ)** that are out of date:

```
factor: $(OBJ)
        cc -c $? -lm
```

mmacros also contains a default command that describes how to build additional kinds of files:

- **AS** and **ASFLAGS** call the *assembler* to assemble **.obj** files out of source modules written in assembly language rather than C.

You can change the default rules of **make** by changing them in **mactions** and changing the definition of any of the macros as given in **mmacros**.

Double-colon target lines

An alternative form of target line simplifies the task of maintaining libraries. This form uses the double colon "::" instead of a single colon ':' to separate the name of the target from those of the files on which it depends.

A target name can appear on only one single-colon target line, whereas it can appear on several double-colon target lines. The advantage of using the double-colon target lines is that **make** will remake the target by executing the commands (or its default commands) for the *first* such target line for which the target is older than a file on which it depends.

For example, for the program **factor.exe** described earlier, assume that two versions of the source modules **factor.c** and **atod.c** exist: **factora.c** plus **atoda.c**, and **factorb.c** plus **atodb.c** The **makefile** would appear as follows:

```
OBJ1 = factora.obj atoda.obj
OBJ2 = factorb.obj atodb.obj

factor.exe :: $(OBJ1)
        cc -c $(OBJ1) -lm

factor.exe :: $(OBJ2)
        cc -c $(OBJ2) -lm
```

This **makefile** tells **make** to do the following: (1) Check if either **factora.obj** or **atoda.obj** is younger than **factor.exe**. (2) If either one is, regenerate **factor.exe** using this version of these files. (3) If neither **factora.obj** nor **atoda.obj** is younger than **factor.exe**, then check to see if either **factorb.obj** or **atodb.obj** is younger than **factor.exe**. (4) If either of them is, then regenerate **factor.exe** using the youngest version of these files.

Let's C

This technique allows you to maintain multiple versions of source files in the same directory and selectively recompile the most recently updated version without having to edit your **makefile** or otherwise trick the system.

You cannot target a file in both a single-colon and a double-colon target line.

Alternative uses

make is a program that helps you construct complex things from a number of simpler things.

make usually is used to build complex C programs: the executable file is made from object modules, which are made from source modules and header files. However, **make** can be used to create any type of file that is constructed from one or more source modules. For example, an accountant can use **make** to generate monthly reports from daily inventories: all the accountant has to do is prepare a **makefile** that describes the dependencies (that is, the name of the monthly report they wish to create and the names of the daily inventories from which it is created), and the command required to generate the monthly report. Thereafter, to recreate the report, all the accountant has to do to generate a monthly report is type **make**.

In another example, the **makefile** can trigger program maintenance commands. For example, the target name **backup** might define commands to copy source modules to another directory; typing **make backup** saves a copy of the source modules. Similar uses include removing temporary files, building libraries, executing test suites, and printing listings. A **makefile** is a convenient place to keep all the commands used to maintain a program.

The following example shows a **makefile** that defines two special target files, **printall** and **printnew**, to be used with the source files for the program **factor.exe**.

```
# This makefile generates the program "factor.exe".
# "factor.exe" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# libm, but it requires no special header files.

OBJ = factor.obj atod.obj
SRC = factor.c atod.c

factor: $(OBJ)
        cc -o factor $(OBJ) -lm

# program to print all the updated source modules
# used to generate the program "factor.exe"


printall:
        pr $(SRC) | print /p
        echo junk > printall

printnew: $(OBJ)
        pr $? | print /p
        echo junk > printnew
```

In this instance, typing the command

```
        make printall
```

forces **make** to generate the target **printall** rather than the target **factor.exe**, which is the default as it appears first in the **makefile**. The **pr** and **print** commands are then used to print a listing of all files defined by **SRC**. The macro **OBJ** cannot be used with these commands because it would trigger the printing of the object files, which would not be of much use. The word **junk** is echoed into an empty file, **prnew**. This new file serves only to record the time the listing is printed. This tactic is performed in order to record the time that the listing was last generated so that **make** will know what files have been updated when you next use **printnew**.

Typing the command

```
make printnew
```

forces **make** to generate the target **printnew** rather than the default target **factor**. **printnew** prints only the files named in the macro **SRC** that have changed since any files were last printed.

Special targets

A few target names have special meanings to **make**. The name of each special target begins with '.' and contains upper-case letters.

The target name **.DEFAULT** defines the default commands **make** uses if it cannot find any other way to build a target. The special target **.IGNORE** in a **makefile** has the same effect as the **-i** command line option. Similarly, **.SILENT** has the same effect as the **-s** command line option.

Errors

make prints "*command* exited with status *n*" and exits if an executed *command* returns an error status. However, it ignores the error status and continues processing if the **makefile** command line begins with a hyphen '**-**' or if the **make** command line specifies the **-i** option.

make reports an error status and exits if the user interrupts it. It prints "**can't open** *file*" if it cannot find the specification *file*. It prints "**Target** *file* **is not defined**" or "**Don't know how to make** *target*" if it cannot find an appropriate *file* or commands to generate *target*. Other possible errors include syntax errors in the specification file, macro definition errors, and running out of space. The error messages **make** prints are generally self-explanatory; however, a table of error messages and brief descriptions of them are given in a later section of this manual.

Exit status

make returns a status of zero if it succeeds and -1 if an error occurs.

Where to go from here

make is summarized in the Lexicon. Look there for more information about how to use it with C programs.

Questions and Answers

The following is a list of questions asked most often by **Let's C** users, and suggestions for solving problems. If you have a problem with **Let's C**, look here first.

Programming problems

Why doesn't **cpp** *execute?* **cpp** *execute?*>=29

Most likely, **cc** cannot find **cpp** because it is in another directory and you did not tell **cc** where to look for it. If this is the case, use the **-xc** option in the **cc** command line, as described in the Lexicon entry for **cc**. Also see the sub-section *Setting the environment*, in section 1 of this manual.

Can I keep the compiler and source code on separate disks?

Yes, when you use the **-x** options on the **cc** command line. See the description of these options in the Lexicon entry for **cc**. Also see the sub-section *Setting the environment*, in section 1 of this manual.

My program won't read a carriage return from a file. Why? read a carriage return from a file. Why?'>=29

When you open a file stream, by default it is opened in ASCII mode. A file stream opened in ASCII mode will handle only alphanumeric characters plus the newline character '\n'. All other characters, including the carriage return character '\r' will be dropped from the file stream. To read a file that contains a carriage return or other non-alphanumeric characters correctly, open the file in binary mode.

To read from a binary file you must open it in binary mode; for example:

```
fopen("filename", "rb")
```

For more information, see the entry for **fopen** in the Lexicon.

My automatic large array is corrupted. Why?

Most likely, you did not allocate enough stack when you compiled your program. **Let's C** by default sets aside two kilobytes of memory for stack, but your program may require more. To increase stack size, use the **-ys** option to **cc**, or make the array static by moving it outside of the body of the program. The **-ys** option takes the number of bytes in decimal; for example, **-ys 10000** gives you 10,000 bytes worth of stack.
Note that using too much stack space can itself cause other unpredictable results to appear.

Can I reduce the size of my compiled modules?

Yes. A number of techniques will save space in your programs. For example, try making automatic variables into register variables. This also increases the speed of execution. Use register variables only for heavily used data items. Normally, the compiler uses registers SI and DI for intermediate work. The first register variable you assign uses SI, the second uses DI. After that, the **register** typing is ignored.

Another technique is to perform repetitive function calls by means of a table definition. This will eliminate the code needed to make each of the function calls except one. Also, try removing the modules' symbol tables with the command **strip**. This will reduce their size significantly.

If your program does not use any STDIO routines, you can compile it with the **-ns** option.

The order of evaluation is not what I expected. Why?

Note the following passage from *The C Programming Language*: "C, like most languages, does not specify in what order the operands of an operator are evaluated .... In any expression that involves side effects, there can be subtle dependencies on the order in which variables taking part in the expression are stored. One unhappy situation is typified by the statement

```
a[i] = i++;
```

"The question is whether the subscript is the old value of **i** or the new. The compiler can do this in different ways, and generate different answers depending on its interpretation... The moral of this discussion is that writing code which depends on order of evaluation is a bad programming practice in any language."

printf *gives incorrect output when rounding numbers. Why?*

For example:

```
printf("%6.0f", (double) 9/10.0);
```

yields 0 instead of 1. This, however, represents a misunderstanding of how **printf** works. The instruction **%6.0f** tells **printf** to truncate the value and print it, not round it; because 9/10 is less than 1, the output is zero, not one.

Where does **make** *look for* **mactions** *and* **mmacros**?

mmacros and **mactions** are files of preset macros and definitions that **make** uses by default. **make** looks for them first in the directories named by the environmental variable **LIBPATH**. If this variable is not set, it then looks in directory **lib**; if there is no directory with this name, it finally looks in the current directory.

My Tandy 2000 cannot access my printer with **lpt1**. *Why?*

The Tandy 2000 is not fully IBM-compatible. One way in which it differs from the IBM PC is that it cannot recognize the logical device **lpt1**. Use **prn** to access the printer on this machine.

What does the error message **temporary file write error** *mean?*

Let's C is a multiphase compiler in which each phase performs a different task. Because each phase stands alone (as a single program), it must write its output to a temporary file that is read by the following phase. Thus, this error means that a phase could not write out its temporary file. This is usually due to a hardware problem such as a full disk, or a write-protected disk. To overcome this problem, it might be necessary to write temporary files to another directory or to another disk drive.

What does the error message **lvalue required** *mean?*

Your program uses a constant where it should use a variable. A *variable* is the name of any data element whose value can change; for example

```
int foo;
```

declares the variable **foo**. A *constant*, on the other hand, is any number or fixed address. The name of an array, for example, is a fixed address and cannot be altered. The code

```
int foo[];
int *bar;
   ...
foo = bar;
```

will generate an **lvalue required** error message, because the name of an array is a constant rather than a variable. On the other hand, the code

```
int *foo;
int *bar;
   ...
foo = bar;
```

will not, because both **foo** and **bar** are pointers and, therefore, are variables. See the Lexicon entries for **lvalue** and **rvalue** for more information.

What does the message **Identifier** *string* **is being redeclared** *mean?*

If you use a function without declaring it, **Let's C** assumes that it is an integer. If later in your program you declare that function to be something other than an integer, your declaration will clash with the implicit declaration you made earlier, and so trigger the error message. You should check that your functions do not contradict themselves. It is a good programing practice to declare explicitly all functions and variables your program will use.

What does the error message **out of space** *mean?*

Most likely, you created a function that is too large for the compiler to process. Break the routine into smaller components.

If this error is generated by **cc2**, try recompiling your program with the option **-vcc2l**. This tells **cc** to use a LARGE-model version of **cc2**; this version runs more slowly than the normal SMALL-model version of **cc2**, but can handle larger programs.

What does **Bad value in debug** *mean?*

This error occurs only when you have used the **-VCSD** option, so that you can debug your program with **csd**. Your program probably declared a pointer to a structure tag that does not exist. Check the declaration of the structure or pointer.

How can I estimate how much stack I need?

Automatic variables and passed parameters go on the stack. Register variables do not go onto the stack. Each level of call requires eight bytes in SMALL model, and 10 bytes in LARGE model. The runtime startup routine needs about 200 bytes of stack, and **printf** about 100 bytes.

The default stack size is two kilobytes (2,048 bytes). To change it, use the **-ys** option; for example, to compile a program with 4,096 bytes (four kilobytes) of stack, use the following command:

```
cc -ys 4096 example.c
```

How do **execall** *and* **system** *differ?*

execall sends a command and its list of arguments, or "tail", directly to MS-DOS; **system**, on the other hand, sends a command through **command.com**.

execall looks for the executable file, loads it, executes it with the given tail as its arguments, and returns its exit status code. Thus, it only works if command exits to its caller rather than by executing the MS-DOS warm boot. MS-DOS built-in commands, such as **dir**, do not work with **execall** for this reason. **system** passes a command line to **command.com**, loads it, and executes it as if it had been typed at the MS-DOS command level. **system** can be used with the MS-DOS built-in commands, as well as with commands that rely on MS-DOS to parse the command line into the formatted parameter area. Note, too, that **system** runs more slowly than **execall**, and it cannot pass to the calling program what the called program returned upon exiting.

See the Lexicon entries for **execall** and **system** for more information and for example programs that use these routines.

What does the runtime startup routine do?

This is a routine that is linked with a C program as the first part of the executable object program. It initializes the stack and saves information necessary to return to the calling program, and calls the C library function **_main** to parse the MS-DOS command tail into the arguments **argv**, **argc**, and **envp**, which are expected by the C program.

How can I make ROMable code?

Use the following steps:

1. Use the option **-VROM** to move constant strings into the code segment.
2. Put the data segment into ROM, copy it to RAM, and have the data segment point to where it is in RAM. This must be done explicitly, i.e., you must write a new runtime startup routine.
3. STDIO routines are linked into a program even if they are not required. Use the **cc** option **-ns** to exclude them from your program. This also gives the program a different version of the **exit** command, which does not call **fflush** or **fclose**.
4. Tools for converting to Intel hex format and for burning PROMs must be purchased from third-party vendors. Mark Williams Company does not supply them at present.

How can I declare an array of (row)(col) elements?*

Declare and initialize it to **Array[row-1][col-1]**. The first element of the array is **Array[0][0]**.

How can I redirect error messages into a file?

Use the greater-than sign '>' with MS-DOS. For example,

```
cc filename.c > errfile
```

will work for one file. For multiple compiles, say:

```
cc file1.c > errfile
cc file2.c >> errfile
cc file3.c >> errfile
```

This appends the error messages from subsequent compilations onto the error file.

The **-A** option to **cc** automatically redirects error messages into a buffer, and invokes the MicroEMACS editor so you can fix your source file "on the spot". You may find this to be more convenient than redirecting the error messages into a file. For more information on this option, see the Lexicon entry for **cc**.

How can I redirect an object file to another directory?

The option **-o** *filename* redirects the object file into *filename*, whereas the option **-xo** *directory* redirects it into *directory*.

How can I build pointers for segment and offset functions, like **copy***?*

Use the function **ptoreg** to convert C pointers to processor register pairs. **ptoreg** converts a pointer *p* relative to segment *seg* and stores the resulting segment:offset pair in the register pair *segreg:offreg*.

The functions **csreg**, **dsreg**, **esreg**, and **ssreg** return the current segment register values. To turn a register pair into a C pointer, use the function **regtop**.

Can I compile a program from within MicroEMACS?

Yes. Use the -A option to the **cc** command line. If an error occurs, you will be returned to MicroEMACS automatically, with your source code displayed in one window and the compiler's error messages displayed in the other. When you have corrected the problem, exiting from the editor with either the **<ctrl-X><ctrl-S>** or **<ctrl-Z>** automatically recompiles your program.

Let's C

Problems with running programs

My data are being corrupted inexplicably.

My computer is hanging.

My program is generating garbage.

These problems may have a number of causes; the most likely is improper allocation of space. Often, the stack size is too small. If the stack grows too large for the space that has been allocated for it, it will invade and corrupt the static data area. The default value for stack size is two kilobytes (2,048 bytes), which is large enough for most functions; however, but highly recursive functions (such as **qsort**) or programs that use large automatic arrays (such as the sample program on page 29 of *The C Programming Language,* ed. 2) will quickly exhaust the available stack space. There is no way to increase the size of the stack while a program is running. To increase the stack, you must relink the program and allocate more stack by using the **-ys** option to the **cc** command. For more information, see the Lexicon entries for **cc** and **stack**.

Another common cause of data corruption is using a pointer without allocating space for the object to which it points. This is called an *uninitialized pointer*. For instance, if your program declares the variable **str** to be a pointer to a **char**, you cannot assign data to **str** unless you ensure that **str** points to a place that can hold these data; otherwise, the results will be unpredictable. You can make sure that a pointer works correctly either by **initializing** it, or by allocating space with **malloc** or **calloc**.

Another cause of this problem is passing a function the wrong number or type of parameters. Be sure that all functions have the correct number of arguments, and that all arguments are of the correct type.

My output is not going to the screen as I expected.

MS-DOS buffers output to the console, and does not print it until it gets a newline character. You can flush out the buffer whenever you want by using the function call **fflush(stdout)**, or you can use the Mark Williams functions **getcnb** and **putcnb**, which go directly to the console.

getcnb *doesn't work right. Why?* work right. Why?'>=29

Problems will arise when you combine **getcnb** with **printf** or any other normal STDIO function. **Let's C** follows the UNIX protocol, and buffers all of its STDIO functions. For example, when you create a **printf** string, it waits in a buffer until something, such as a newline character or a **fflush** instruction, pushes it out of the buffer and onto your screen. Thus, if you use a **printf** call to print a prompt string, then use a **getcnb** call to get the user's response, you will not see the prompt until you type the carriage return in response to the **getcnb** call. To solve this problem, either use **putcnb** to display the prompt, or follow the **printf** call with **fflush(stdout)**.

How do I clear the screen?

The easiest way to do this is to use the appropriate escape sequences defined in the file **ansi.sys**, provided it is loaded by **config.sys**. You can also call the MS-DOS function that clears the screen. See the Lexicon entry for **ansi.sys** for more information.

Can I open more than the default number of files at a time?

Yes. Simply insert the instruction

```
FILES=n
```

into the file **config.sys**, where *n* is the number of files you want to be able to open at any given time. Because of the way MS-DOS is designed, no more than 20 files can be opened by a program at any one time; this limit *includes* **stdin**, **stdout**, **stderr**, **aux**, and the

printer. Make sure that **config.sys** is on your boot disk, and then reboot your system.

Can I call any MS-DOS function or interrupt?

The function **intcall**, which is described in the Lexicon, provides a general interrupt calling routine. See the Lexicon entry for **interrupts** for the number of the interrupt you need to hand to **intcall**, the number of the function you wish to call, and any other information the function requires. Also read the header file **dos.h**, which defines constants for most of the interrupts and function numbers.

For a summary of how to handle interrupts, see the Lexicon entry for **interrupt handling**.

How can I position the cursor?

The easiest way is to use the escape sequences listed in the file **ansi.sys**. See the Lexicon entry for **ansi.sys** for more information. MS-DOS interrupt 10 can also be used to move the cursor. The MicroEMACS editor uses interrupt 10, and its source code (which is included with **Let's C**) demonstrates how to use this interrupt.

Can I link my **masm** *routines with* **Let's C** *output?*

Yes, as long as you observe **Let's C**'s linkage conventions. For more information, see the Lexicon entry on **calling conventions**.

The command **fixobj** lets you edit object modules. With **fixobj**, you can edit modules compiled or assembled by other language tools so that they can be linked with programs generated by **Let's C**. For more information, see the Lexicon entry for **fixobj**.

Where does **Let's C** *put things in memory?*

See the entry on **memory allocation** in the Lexicon.

Limitations in i8086

What are the limits on the size of arrays?

Let's C does not limit the size of an array; however, the architecture of the Intel i8086 microprocessor is such that it forbids the creation of a data structure that is larger than 64 kilobytes.

SMALL-model limitations

Programs are limited to 128 kilobytes of code and data combined. Within the 128 kilobytes, the following limitations apply:

- No program can have more than 64 kilobytes of code.
- No program can have more than 64 kilobytes of data.

Data includes stack (automatic) data, static data, and dynamically allocated memory.

LARGE-model limitations

Programs are limited to one megabyte of code and data combined. Within the one megabyte, the following limitations apply:

- No module can have more than 64 kilobytes of code.
- No module or library can have more than 64 kilobytes of static data.
- The stack size cannot exceed 64 kilobytes.
- No individual data structure can exceed 64 kilobytes.

Error Messages

This chapter lists all of the error messages that can be produced by the compiler, the assembler **as**, and **make**.

The messages are in alphabetical order, and each is marked to indicate which program generated it (e.g., **cc0**, **ccp**). Each message from the compiler indicates whether it is a *fatal*, *error*, *warning*, or *strict* condition. The compilation phases are **cpp**, the preprocessor; **cc0**, the parser; **cc1**, the code generator; **cc2**, the optimizer; and **cc3**, the disassembler.

A fatal message usually indicates a condition that caused the compiler to terminate execution. Fatal errors from the later phases of compilation often cannot be fixed, and may indicate problems in the compiler.

An error message points to a condition in the source code that **Let's C** cannot resolve. This almost always occurs when the program does something illegal, e.g., has unbalanced braces.

Warning messages point out code that is compilable, but may produce trouble when the program is executed. A strict message refers to a passage in the code that is unorthodox and may not be portable.

add of two non-constants (**as**, error)
: The present expression adds one or more elements that will be relocated.

address out of range (**as**, error)
: **jmp** addresses must be to a place within 127 bytes.

address wraparound (**ld**, fatal)
: A segment of the program has exceeded the size allowed by the microprocessor's architecture.

; after target or macroname (**make**, error)
: A semicolon appeared after a target name or a macro name.

ambiguous reference to "*string*" (**cc0**, error)
: *string* is defined as a member of more than one **struct** or **union**, is referenced via a pointer to one of those **struct**s or **union**s, and there is more than one offset that could be assigned.

argument list has incorrect syntax (**cc0**, error)
: The argument list of a function declaration contains something other than a comma-separated list of formal parameters.

string argument mismatch (**cpp**, error)
: The argument *string* does not match the type declared in the function's prototype. Either the function prototype or the argument should be changed.

array bound must be a constant (**cc0**, error)
: An array's size can be declared only with a constant; you cannot declare an array's size by using a variable. For example, it is correct to say **foo[5]**, but illegal to say

```
bar = 5;
foo[bar];
```

array bound must be positive (**cc0**, error)
: An array must be declared to have a positive number of elements. The array flagged here was declared to have a negative size, e.g., **foo[-5]**.

array bound too large (**cc0**, error)
: The array is too large to be compiled with 16-bit index arithmetic. You should devise a way to divide the array into compilable portions.

array row has 0 length (**cc0**, error)
: This message can be triggered by either of two problems. The first problem is declaring an array to have a length of zero; e.g., **foo[0]**. The second problem is failing to declare the size of a dimension *other than the first* in a multi-dimensional array. C allows you to declare an indefinite number of array elements of *n* bytes each, but you cannot declare *n* array elements of an indefinite length. For example, it is correct say **foo[][5]** but illegal to say **foo[5][]**.

#assert failure (**cpp**, error)
: The condition being tested in a **#assert** statement has failed.

associative expression too complex (**cc1**, fatal)
: An expression that uses associative binary operators (e.g., '+') has too many operators; for example, **i=i1+i2+i3+ . . . +i30;**. You should simplify the expression.

at beginning of macro (**cpp**, error)
: Macro replacement lists may contain tokens that are separated by **##**, but **##** cannot appear at the beginning or the end of the list. The tokens on either side of the **##** are pasted together into one token.

at end of macro (**cpp**, error)
: Macro replacement lists may contain tokens that are separated by **##**, but **##** cannot appear at the beginning or the end of the list. The tokens on either side of the **##** are pasted together into one token.

auto "*string*" is not addressable (**cc0**, error)
: The identifier *string* cannot be addressed on the stack, probably because it is an extraordinarily large automatic array. Large automatic arrays should usually be declared global or static, not automatic.

bad argument storage class (**cc0**, error)
: An argument was assigned a storage class that the compiler does not recognize. The only valid storage class is **register**.

bad base type for field (**cc0**, error)
: The expression uses a bitwise operator (i.e., '<<', '>>', '&', or '|') with an incorrect type of variable. A bit field must be declared within a **char**, **unsigned char**, **int**, or **unsigned int**. No other base type is allowed.

bad external storage class (**cc0**, error)
: An **extern** has been declared with an invalid storage class, e.g., **register** or **auto**.

bad field width (**cc0**, error)
: A field width was declared either to be negative or to be larger than the object that holds it. For example, **char foo:9** or **char foo:-1** will trigger this error.

bad filler field width (**cc0**, error)
: A filler field width was declared either to be negative or to be larger than the object that holds it. For example, **char foo:9** or **char foo:-1** will trigger this error.

bad flexible array declaration (**cc0**, error)
: A flexible array is missing an array boundary; e.g., **foo[5][]**. C permits you to declare an indefinite number of array elements of *n* bytes each, but you cannot declare an array to have *n* elements of an indefinite number of bytes each.

bad line number after # (**as**, error)
The present expression uses an incorrect line number after a '#', e.g,. a negative number.

Bad macro name (**make**, error)
A bad macro name was used; for example, a macro name included a control character.

break not in a loop (**cc0**, error)
A **break** occurs that is not inside a loop or a **switch** statement.

call of non function (**cc0**, error)
What the program attempted to call is not a function. Check to make sure that you have not accidentally declared a function as a variable; e.g., typing **char *foo;** when you meant **char *foo();**.

cannot add pointers (**cc0**, error)
The program attempted to add two pointers. **int**s or **long**s may be added to or subtracted from pointers, and two pointers to the same type may be subtracted, but no other arithmetic operations are legal on pointers.

cannot apply unary '&' to a bit field (**cc0**, error)
The program attempted to use the address of a bit within a byte, which is illegal. Only bytes can be addressed, not the bits within them.

cannot apply unary '&' to a register variable (**cc0**, error)
Because register variables are stored within registers, they do not have addresses, which means that the unary **&** operator cannot be used with them.

cannot apply unary '&' to an alien function (**cc0**, error)
The unary '&' operator cannot be used with any function that has been declared to be of type **alien**. **alien** functions cannot be called by pointers.

cannot cast double to pointer (**cc0**, error)
The program attempted to cast a **double** to a pointer. This is illegal.

cannot cast pointer to double (**cc0**, error)
The program attempted to cast a pointer to a **double**. This is illegal.

cannot cast structure or union (**cc0**, error)
The program attempted to cast a **struct** or a **union**. This is illegal.

cannot cast to structure or union (**cc0**, error)
The program attempted to cast a variable to a **union** or **struct**. This is illegal.

string: cannot create (**as**, error)
The assembler cannot create the output file it was requested to create. This often is due to a problem with the output device; check and make sure that it is not full, and that it is working correctly.

string: cannot create (**cpp**, fatal)
The preprocessor **cpp** cannot create the output file *string* that it was asked to create. This often is due to a problem with the output device; check and make sure that it is not full and that it is working correctly.

cannot declare array of functions (**cc0**, error)
For example, the declaration **extern int (*f)[]();** declares **f** to be an array of pointers to functions that return **int**s. Arrays of functions are illegal.

cannot declare flexible automatic array (**cc0**, error)
The program does not explicitly declare the number of elements in an automatic array.

cannot fold this expression (**as**, error)
: The assembler cannot fold an expression, e.g., **(ax+bx)**. This can occur for any of several reasons.

cannot initialize fields (**cc0**, error)
: The program attempted to initialize bit fields within a structure. This is not supported.

cannot initialize unions (**cc0**, error)
: The program attempted to initialize a **union** within its declaration. **union**s cannot be initialized in this way.

cannot move '.' back (**as**, error)
: The assembler cannot move the location counter backward, only forward.

string: cannot open (**cpp**, **cc0**, fatal)
: The compiler cannot open the file *string* of source code that it was asked to read. **cpp** may not have been told the correct directory in which this file is to be found; check that the file is located correctly, and that the **-I** options, if any, are correct.

cannot open include file *string* (**cpp**, **cc0**, fatal)
: The program asked for file *string*, which was not found in the same directory as the source file, nor in the default **include** directory specified by the environmental variable **INCDIR**, nor in any of the directories named in **-I** options given to the **cc** command.

string: cannot reopen (**cc2**, fatal)
: The optimizer in **cc2** cannot reopen a file with which it has worked. Make sure that your mass storage device is working correctly and that it is not full.

case not in a switch (**cc0**, error)
: The program uses a **case** label outside of a **switch** statement. See the Lexicon entry for **case**.

character constant overflows long (**cc0**, error)
: The character constant is too large to fit into a **long**. It should be redefined.

character constant promoted to long (**cc0**, warning)
: A character constant has been promoted to a **long**.

class not allowed in structure body (**cc0**, error)
: A storage class such as **register** or **auto** was specified within a structure.

compound statement required (**cc0**, error)
: A construction that requires a compound statement does not have one, e.g., a function definition, array initialization, or **switch** statement.

conditional stack overflow (**cpp**, fatal)
: A series of **#if** expressions is nested so deeply that it overflowed the allotted stack space. You should simplify this code.

constant expression required (**cc0**, error)
: The expression used with a **#if** statement cannot be evaluated to a numeric constant. It probably uses a variable in a statement rather than a constant.

constant "*number*" promoted to long (**cc0**, warning)
: The compiler promoted a constant in your program to **long**; although this is not strictly illegal, it may create problems when you attempt to port your code to another system, especially if the constant appears in an argument list.

constant used in truth context (**cc0**, strict)
A conditional expression for an **if**, **while**, or **for** statement has turned out to be always true or always false. For example, **while(1)** will trigger this message.

construction not in Kernighan and Ritchie (**cc0**, strict)
This construction is not found in *The C Programming Language*; although it can be compiled by **Let's C**, it may not be portable to another compiler.

continue not in a loop (**cc0**, error)
The program uses a **continue** statement that is not inside a **for** for **while** loop.

data in bssd (**as**, error)
The program attempted to initialize something in the **bssd** segment, which can contain only uninitialized data.

'.' declared as label (**as**, error)
The present expression uses as a label, e.g., ".:". This is illegal.

#define argument mismatch (**cpp**, warning)
The definition of an argument in a **#define** statement does not match its subsequent use. One or the other should be changed.

declarator syntax (**cc0**, error)
The program used incorrect syntax in a declaration.

default label not in a switch (**cc0**, error)
The program used a **default** label outside a **switch** construct. See the Lexicon entry for **default**.

divide by zero (**cc0**, warning)
The program will divide by zero if this code is executed. Although the program can be parsed, this statement may create trouble if executed.

duplicated case constant (**cc0**, error)
A **case** value can appear only once in a **switch** statement. See the Lexicon entries for **case** and **switch**.

#elif used without #if or #ifdef (**cpp**, error)
An **#elif** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

#elif used after #else (**cpp**, error)
An **#elif** control line cannot be preceded by an **#else** control line.

#else used without #if or #ifdef (**cpp**, error)
An **#else** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

empty switch (**cc0**, warning)
A **switch** statement has no **case** labels and no **default** labels. See the Lexicon entry for **switch**.

#endif used without #if or #ifdef (**cpp**, error)
An **#endif** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

EOF in comment (**cpp**, fatal)
Your source file appears to end in mid-comment. The file of source code may have been truncated, or you failed to close a comment; make sure that each open-comment symbol '/*' is balanced with a close-comment symbol '*/'. Also, be sure that you did not accidentally embed a **<ctrl-Z>** in the line.

EOF in macro *string* invocation (**cpp**, error)
Your source file appears to end in a macro call. The source file may have been truncated, or you may have accidentally embedded a **<ctrl-Z>** in the line.

EOF in midline (**cpp**, warning)
Check to see that your source file has not been truncated accidentally. Also, make sure that you did not accidentally embed a **<ctrl-Z>** in the line.

EOF in string (**cpp** ,error)
Your file appears to end in the middle of a quoted string literal. Check to see that your source file has not been truncated accidentally. Also, check that you did not accidentally embed a **<ctrl-Z>** in the line.

#error: *string* (**cpp**, fatal)
An **#error** control line has been expanded, printing the remaining tokens on the line and terminating the program.

error creating address (**as**, error)
The object generator could not build an address in MS-DOS object format. Please contact Mark Williams Company.

error in #define syntax (**cpp**, error)
The syntax of a **#define** statement is incorrect. See the Lexicon entry for **#define** for more information.

error in enumeration list syntax (**cc0**, error)
The syntax of an enumeration declaration contains an error.

error in expression syntax (**cc0**, error)
The parser expected to see a valid expression, but did not find one.

error in #include syntax (**cpp**, error)
An **#include** directive must be followed by a string enclosed by either quotation marks (" ") or angle brackets (<>). Anything else is illegal.

expected comma (**as**, error)
The assembler expected to find a comma in the present expression, but did not.

expected constant (**as**, error)
The assembler expected to find a constant in the present expression, but did not.

exponent overflow in floating point constant (**cc0**, warning)
The exponent in a floating point constant has overflowed. The compiler has set the constant to the maximum allowable value, with the expected sign.

exponent underflow in floating point constant (**cc0**, warning)
The exponent in a floating point constant has underflowed. The compiler has set the constant to zero, with the expected sign.

expression too complex (**cc1**, fatal)
The code generator cannot generate code for an expression. You should simplify your code.

external syntax (**cc0**, error)
This could be one of several errors, most often a missing '{'.

field too wide (**cc0**, error)
A field must fit within an **int**, so the declared field width must not be greater than 16.

file ends within a comment (**cc0**, error)
The source file ended in the middle of a comment. If the program uses nested comments, it may have mismatched numbers of begin-comment and end-comment markers. If not, the

program began a comment and did not end it, perhaps inadvertently when dividing by *something, e.g., **a=b/*cd;**.

function cannot return a function (**cc0**, error)
: The function is declared to return another function, which is illegal. A function, however, can return a *pointer* to a function, e.g., **int (*signal(n, a))()**

function cannot return an array (**cc0**, error)
: A function is declared to return an array, which is illegal. A function, however, can return a pointer to a structure or array.

functions cannot be parameters (**cc0**, error)
: The program uses a function as a parameter, e.g., **int q(); x(q);**. This is illegal.

identifier *string* has too many arguments (**cpp**, error)
: Too many actual parameters have been provided.

identifier "*string*" is being redeclared (**cc0**, error)
: The program declares variable *string* to be of two different types. This often is due to an implicit declaration, which occurs when a function is used before it is explicitly declared. Check for name conflicts.

identifier "*string*" is not a label (**cc0**, error)
: The program attempts to **goto** a nonexistent label.

identifier "*string*" is not a parameter (**cc0**, error)
: The variable "*string*" did not appear in the parameter list.

identifier "*string*" is not defined (**cc0**, error)
: The program uses identifier *string* but does not define it.

identifier "*string*" not bound to register (**cc0**, strict)
: **Let's C** allows two variables to be bound to registers. If more than two variables are declared to be of type **register**, the first two will be bound to registers and all others defined as ordinary **auto**s. If a variable is declared **register** but does not fit in a 16-bit register, it is defined as an **auto**.

identifier "*string*" not usable (**cc0**, error)
: *string* is probably a member of a structure or **union** which appears by itself in an expression.

illegal character constant (**cc0**, error)
: A legal character constant consists of a a backslash '\' followed by **a**, **b**, **f**, **n**, **r**, **t**, **v**, **x**, or up to three octal digits.

illegal character (*number* decimal) (**cc0**, error)
: A control character was embedded within the source code. *number* is the decimal value of the character.

illegal # construct (**cc0**, error)
: The parser recognizes control lines of the form **#***line_number* (decimal) or **#***file_name*. Anything else is illegal.

illegal control line (**cpp**, error)
: A '#' is followed by a word that the compiler does not recognize.

illegal cpp character (*n* decimal) (**cpp**, error)
: The character noted cannot be processed by **cpp**. It may be a control character or a non-ASCII character.

illegal integer constant suffix (**cc0**, error)
: Integer constants may be suffixed with **u**, **U**, **l**, or **L** to indicate **unsigned**, **long**, or **unsigned long**.

illegal label "*string*" (**cc0**, error)
: The program uses the keyword *string* as a **goto** label. Remember that each label must end with a colon.

illegal operation on "void" type (**cc0**, error)
: The program tried to manipulate a value returned by a function that had been declared to be of type **void**.

illegal structure assignment (**cc0**, error)
: The structures have different sizes.

illegal subtraction of pointers (**cc0**, error)
: A pointer can be subtracted from another pointer only if both point to objects of the same size.

illegal use of a pointer (**cc0**, error)
: A pointer was used illegally, e.g., multiplied, divided, or &-ed. You may get the result you want if you cast the pointer to a **long**.

illegal use of a structure or union (**cc0**, error)
: You may take the address of a **struct**, access one of its members, assign it to another structure, pass it as an argument, and return. All else is illegal.

illegal use of defined (**cpp**, error)
: The construction **defined(***token***)** or **defined** *token* is legal only in **#if**, **#elif**, or **#assert** expressions.

illegal use of floating point (**cc0**, error)
: A **float** was used illegally, e.g., in a bit-field structure.

illegal use of "void" type (**cc0**, error)
: The program used **void** improperly. Strictly, there are only **void** functions; **Let's C** also supports the cast to **void** of a function call.

illegal use of void type in cast (**cc0**, error)
: The program uses a pointer where it should be using a variable.

improper operand pair (**as**, error)
: The expression uses one or more improper operands to an instruction. For example, **mov a, b** will trigger this message; the instruction should be rendered **mov ax, b** or **mov a, ax**.

string in #if (**cpp**, error)
: A syntax error occurred in a **#if** declaration. *string* describes the error in detail.

= in or after dependency (**make**, error)
: An equal sign '**=**' appeared within or followed the definition of a macro name or target file; for example, **OBJ=atod.obj=factor.obj** will produce this error.

inappropriate signed (**cc0**, error)
: The **signed** modifier may only be applied to **char**, **short**, **int**, or **long** types.

include stack overflow (**cpp**, fatal)
: A set of **#include** statements is nested so deeply that the allotted stack space cannot hold them. Examines the files for a loop. You should try to fold some of the header files into one, instead of having them call each other.

Incomplete line at end of file (**make**, error)
An incomplete line appeared at the end of the **makefile**.

(in|out)(b|) must be DX or constant (**as**, error)
The present expression must use either DX or a constant.

(in|out)(b|) must use AX or AL (**as**, error)
The present expression must use use AX or AL.

inappropriate "alien" modifier (**cc0**, error)
The **alien** type is used to interface C with non-C functions; your program tried to use **alien** as an internal function rather than as a reference to an external function.

inappropriate "long" (**cc0**, error)
Your program used the type **long** inappropriately, e.g., to describe a **char**.

inappropriate "short" (**cc0**, error)
Your program used the type **short** inappropriately, e.g., to describe a **char**.

inappropriate "unsigned" (**cc0**, error)
Your program used the type **unsigned** inappropriately, e.g., to describe a **double**.

index by non-register (**as**, error)
The present expression attempted to index either by using a variable, or by using a non-existent register.

indirection through non pointer (**cc0**, error)
The program attempted to use a scalar (e.g., a **long** or fBint) as a pointer; you must first cast it to a pointer.

initializer too complex (**cc0**, error)
An initializer was too complex to be calculated at compile time. You should simplify the initializer to correct this problem.

integer pointer comparison (**cc0**, strict)
The program compares an integer or **long** with a pointer without casting one to the type of the other. Although this is legal, the comparison may not work on machines with non-integer pointers, e.g., Z8001 or LARGE-model i8086, or on machines with pointers larger than **int**s, e.g., the 68000.

integer pointer pun (**cc0**, strict)
The program assigns a pointer to an integer, or vice versa, without casting the right-hand side of the assignment to the type of the left-hand side. For example,

```
char *foo;
long bar;
foo = bar;
```

Although this is permitted, it is often an error if the integer has less precision than the pointer does, as in LARGE-model programs. Make sure that you properly declare all functions that returns pointers.

internal compiler error (**cc0**, **cc1**, **cc2**, **cc3**, fatal)
The program produced a state that should not happen during compilation. Forward a copy of the program, preferably on a machine-readable medium, to Mark Williams Company, together with the version number of the compiler, the command line used to compile the program, and the system configuration. For immediate advice during business hours, telephone Mark Williams Company.

internal error, c=*number* in expr. (**as**, error)
The assembler has detected a situation that "should not occur". Please send a copy of the source code that triggered this error to Mark Williams Company. For immediate help during business hours, contact Mark Williams Company.

invalid floating-point register (**as**, error)
The present instruction addresses a floating-point register that does not exist.

invalid identifier (**as**, error)
The present expression uses an invalid identifier, e.g., **39xy**.

invalid index (**as**, error)
The present expression attempted to index in a context where it is illegal.

invalid index register (**as**, error)
The present expression attempted to index using an incorrect register, e.g., **cx**, **dx**, **sp**, **ip**, ***s**, ***l**, ***h**.

invalid local symbol (**as**, error)
The present expression uses an invalid local symbol, e.g., **21f**.

invalid operand (**as**, error)
The program uses an invalid operand, e.g., **mov ax, 39f**.

invalid operand pair (**as**, error)
The expression uses one or two incorrect operands. For example, **mov a, b** will trigger this message; this expression should be rendered **mov ax, b** or **mov a, ax**.

invalid operand type (**as**, error)
The present instruction uses an invalid operand type, e.g., **or cs, cs**'.

invalid symbol (**as**, error)
The present instruction uses an invalid symbol, i.e., either an undefined symbol or a symbol that is illegal, such as an opcode or an assembler instruction.

"*string*" is a enum tag (**cc0**, error)

"*string*" is a struct tag (**cc0**, error)

"*string*" is a union tag (**cc0**, error)
string has been previously declared as a tag name for a **struct**, **union**, or **enum**, and is now being declared as another tag. Perhaps the structure declarations have been included twice.

"*string*" is not a tag (**cc0**, error)
A **struct** or **union** with tag *string* is referenced before any such **struct** or **union** is declared. Check your declarations against the reference.

"*string*" is not a typedef name (**cc0**, error)
string was found in a declaration in the position in which the base type of the declaration should have appeared. *string* is not one of the predefined types or a **typedef** name. See the Lexicon entry on **typedef** for more information.

"*string*" is not an "enum" tag (**cc0**, error)
An **enum** with tag *string* is referenced before any such **enum** has been declared. See the Lexicon entry for **enum** for more information.

class "*string*" [*number*] is not used (**cc0**, strict)
Your program declares variable *string* or *number* but does not use it.

jmp must be direct address (**as**, error)
The **jmp** instruction must be used with a direct address; the present expression violates this rule.

label "*string*" undefined (**cc0**, error)
The program does not declare the label *string*, but it is referenced in a **goto** statement.

left side of "*string*" not usable (**cc0**, error)
The left side of the expression *string* should be a pointer, but is not.

lvalue required (**cc0**, error)
The left-hand value of a declaration is missing or incorrect. See the Lexicon entries for **lvalue** and **rvalue**.

macro body too long (**cpp**, fatal)
The size of the macro in question exceeds 200 bytes, which is the limit designed into the preprocessor. Try to shorten or split the macro.

Macro definition too long (**make**, error)
Macro definitions are limited to 128 characters.

macro expansion buffer overflow in *string* (**cpp**, fatal)
A macro call has expanded into more characters than **cpp** can handle. Try to shorten the macro, or break it up.

macro *string* redefined (**cpp**, error)
The program redefined the macro *string*.

macro *string* requires arguments (**cpp**, error)
The macro calls for arguments that the program has not supplied.

macros nested *number* deep, loop likely (**cpp**, error)
Macros call each other *number* times; you may have inadvertently created an infinite loop. Try to simplify the program.

member "*string*" is not addressable (**cc0**, error)
The array *string* has exceeded the machine's addressing capability. Structure members are addressed with 16-bit signed offsets on most machines.

member "*string*" is not defined (**cc0**, error)
The program references a structure member that has not been declared.

mismatched conditional (**cc0**, error)
In a '?:' expression, the colon and all three expressions must be present.

misplaced ":" operator (**cc1**, error)
The program used a colon without a preceding question mark. It may be a misplaced label.

missing "(" (**cc0**, error)
The **if**, **while**, **for**, and **switch** keywords must be followed by parenthesized expressions.

missing ')' (**as**, error)
The assembler expected to find a right parenthesis in the present expression, but did not.

missing ")" (**cc0**, error)
A right parenthesis ')' is missing anywhere after a left parenthesis '('.

missing "=" (**cc0**, warning)
An equal sign is missing from the initialization of a variable declaration. Note that this is a warning, not an error: this allows **Let's C** to compile programs with "old style" initializers, such as **int i 1**. Use of this feature is strongly discouraged, and it will disappear when the

draft ANSI standard for the C language is adopted in full.

missing "," (**cc0**, error)
A comma is missing from an enumeration member list.

missing ":" (**cc0**, error)
A colon ':' is missing after a **case** label, after a default label, or after the '?' in a '?'-':' construction.

missing ";" (**cc0**, error)
A semicolon ';' does not appear after an external data definition or declaration, after a **struct** or **union** member declaration, after an automatic data declaration or definition, after a statement, or in a **for(;;)** statement.

missing ']' (**as**, error)
The assembler expected to find a right bracket in the present expression, but did not.

missing "]" (**cc0**, error)
A right bracket ']' is missing from an array declaration, or from an array reference; for example, **foo[5**.

missing "{" (**cc0**, error)
A left brace '{' is missing after a **struct** *tag*, **union** *tag*, or **enum** *tag* in a definition.

missing "}" (**cc0**, error)
A right brace '}' is missing from a **struct**, **union**, or **enum** definition, from an initialization, or from a compound statement.

missing "while" (**cc0**, error)
A **while** command does not appear after a **do** in a **do**-**while()** statement.

missing #endif (**cpp**, error)
An **#if**, **#ifdef**, or **#ifndef** statement was not closed with an **#endif** statement.

missing label name in goto (**cc0**, error)
A **goto** statement does not have a label.

missing member (**cc0**, error)
A '.' or '->' is not followed by a member name.

missing output file (**cpp**, fatal)
The preprocessor **cpp** found a **-o** option that was not followed by a file name for the output file.

missing right brace (**cc0**, error)
A right brace is missing at end of file. The missing brace probably precedes lines with errors reported earlier.

missing "*string*" (**cc0**, error)
The parser **cc0** expects to see token *string*, but sees something else.

missing semicolon (**cc0**, error)
External declarations should continue with ',' or end with ';'.

missing type in structure body (**cc0**, error)
A structure member declaration has no type.

Multiple actions for *name* (**make**, error)
A target is defined with more than one single-colon target line.

multiple classes (**cc0**, error)
An element has been asigned to more than one storage class, e.g., **extern register**.

Multiple detailed actions for *name* (**make**, error)
A target is defined with more than one single-colon target line.

multiple #else's (**cpp**, error)
An **#if**, **#ifdef**, or **#ifndef** expression can be followed by no more than one **#else** expression.

multiple types (**cc0**, error)
An element has been assigned more than one data type, e.g., **int float**.

multiply defined symbol (**as**, error)
The current line has re-defined or multiply defined a symbol.

must be CL or 1 (**as**, error)
The present expression must use CL or one, but does not.

must load address into register (**as**, error)
For example, **lea y, x** will trigger this message; this expression should be rendered **lea ax x**.

must load direct address (**as**, error)
For example, **lea ax, (bx, si)** will not work on an i8086.

Must use '::' for *name* (**make**, error)
A double-colon target line was followed by a single-colon target line.

nested comment (**cpp**, warning)
The comment introducer sequence '/*' has been detected within a comment. Comments do not nest.

new line in *string* literal (**cpp**, error)
A newline character appears in the middle of a string. If you wish to embed a newline within a string, use the character constant '\n'. If you wish to continue the string on a new line, insert a backslash '\' before the new line.

Newline after target or macroname (**make**, error)
A newline character appears after a target name or a macro name.

newline in macro argument (**cpp**, warning)
A macro argument contains a newline character. This may create trouble when the program is run.

no 'pop CS' instruction (**as**, error)
The assembler expected to find a 'pop CS' instruction in the present expression, but did not.

no string in .ascii statement (**as**, error)
The present expression uses a **.ascii** instruction, but does not give it a string.

non scalar field (**cc0**, error)
A field must be declared within a **char**, **unsigned char**, **int**, or **unsigned int**. A field with an array base type is not allowed.

non-constant in multiply (**as**, error)
The present expression attempts to multiply one or more elements that will be relocated.

non-constant in segment construction (**as**, error)
The present expression attempts to perform a logical OR on an element that will be relocated.

nonterminated string or character constant (**cc0**, error)
: A line that contains single or double quotation marks left off the closing quotation mark. A newline in a string constant may be escaped with '\'.

not a direct address (**as**, error)
: For example, **[(bx)+5]** will trigger this message.

not a direct address in this segment (**as**, error)
: For example, the expression **[x-y]** is acceptable only if **x** and **y** are in the same segment.

'::' not allowed for *name* (**make**, error)
: A double-colon target line was used illegally; for example, after single-colon target line.

number has too many digits (**cc0**, error)
: A number is too big to fit into its type.

only one default label allowed (**cc0**, error)
: The program uses more than one **default** label in a **switch** expression. See the Lexicon entries for **default** and **switch** for more information.

::: or : in or after dependency list (**make**, error)
: A triple colon is meaningless to **make**, and therefore illegal wherever it appears. A single colon may be used only in a target line (which is also called the *dependency list*), and nowhere else.

Out of core (adddep) (**make**, error)
: This results from a system problem. Try reducing the size of your **makefile**.

Out of space (**make**, error)
: System problem. Try reducing the size of your **makefile**.

Out of space (lookup) (**make**, error)
: System problem. Try reducing the size of your **makefile**.

out of space (**cpp**, **cc0**, **cc1**, **cc2**, **cc3**, fatal)
: The compiler ran out of space while attempting to compile the program. To remove this error, examine your source and break up any functions that are extraordinarily large.

out of tree space (**cc0**, fatal)
: The compiler allows a program to use up to 350 tree nodes; the program exceeded that allowance.

output write error (**cc2**, error)
: The optimizer **cc2** cannot create its output file. Check to see if the output device is working correctly, and has enough space to hold the file being created.

parameter *string* is not addressable (**cc0**, error)
: The parameter has a stack frame offset greater than 32,767. Perhaps you should pass a pointer instead of a structure.

parameter must follow # (**cpp**, error)
: Macro replacement lists may contain **#** followed by a macro parameter name. The macro argument is converted to a string literal.

phase error (**as**, error)
: The value of a label changed during the assembly. An expression has a size that differs between the first and second passes.

potentially nonportable structure access (**cc0**, strict)
: A program that uses this construction may not be portable to another compiler.

preprocessor assertion failure (**cpp**, warning)
A **#assert** directive that was tested by the preprocessor **cpp** was found to be false.

string redefined (**cpp**, error)
cpp macros should not be redefined. You should check to see that you are not **#include**ing two different versions of a file somehow, or attempting to use the same macro name for two different purposes.

return type/function type mismatch (**cc0**, error)
What the function was declared to return and what it actually returns do not match, and cannot be made to match.

return(e) illegal in void function (**cc0**, error)
A function that was declared to be type **void** has nevertheless attempted to return a value. Either the declaration or the function should be altered.

risky type in truth context (**cc0**, strict)
The program uses a variable declared to be a pointer, **long**, **unsigned long**, **float**, or **double** as the condition expression in an **if**, **while**, **do**, or '?'-':'. This could be misinterpreted by some C compilers.

segment of improper symbol (**as**, error)
For example, **es:ax** will trigger this message.

segment override by non-segment register (**as**, error)
The present expression uses a non-segment register to set a segment prefix.

size of *string* overflows size_t (**cc0**, strict)
A string was so large that it overran an internal compiler limit. You should try to break the string in question into several small strings.

size of struct "*string*" is not known (**cc0**, error)
small strings.

size of union "*string*" is not known (**cc0**, error)
A pointer to a **struct** or **union** is being incremented, decremented, or subjected to array arithmetic, but the **struct** or **union** has not been defined.

size of *string* too large (**cc0**, error)
The program declared an array or **struct** that is too big to be addressable, e.g., **long a[20000];** on a machine that has a 64-kilobyte limit on data size and four-byte **long**s.

sizeof truncated to unsigned (**cc0**, warning)
An object's **sizeof** value has lost precision when truncated to a **size_t** integer.

sizeof(*string*) set to *number* (**cc0**, warning)
The program attempts to set the value of *string* by applying **sizeof** to a function or an **extern**; the compiler in this instance has set *string* to *number*.

storage class not allowed in cast (**cc0**, error)
The program **cast**s an item as a **register**, **static**, or other storage class.

string initializer not terminated by NUL (**cc0**, warning)
An array of **char**s that was initialized by a string is too small in dimension to hold the terminating NUL character. For example, **char foo[3] = "ABC"**.

structure "*string*" does not contain member "*m*" (**cc0**, error)
The program attempted to address the variable *string.m*, which is not defined as part of the structure *string*.

structure or union used in truth context (**cc0**, error)
The program uses a structure in an **if**, **while**, or **for**, or '?:' statement.

subtracting non-constant (**as**, error)
The present expression subtracts one or more elements that will be relocated.

switch of non integer (**cc0**, error)
The expression in a **switch** statement is not type **int** or **char**. You should cast the **switch** expression to an **int** if the loss of precision is not critical.

switch overflow (**cc1**, fatal)
The program has more than ten nested **switch**es.

symbol "*string*" truncated to 39 characters (**cc2**, warning)
A symbol name can have no more than 39 characters.

Syntax error (**make**, error)
The syntax of a line is faulty.

too many adjectives (**cc0**, error)
A variable's type was described with too many of **long**, **short**, or **unsigned**.

too many arguments (**cc0**, fatal)
No function may have more than 30 arguments.

too many arguments in a macro (**cpp**, fatal)
The program uses more than the allowed ten arguments with a macro.

too many cases (**cc1**, fatal)
The program cannot allocate space to build a **switch** statement.

too many directories in include list (**cpp**, fatal)
The program uses more than the allowed ten **#include** directories.

too many initializers (**cc0**, error)
The program has more initializers than the space allocated can hold.

Too many macro definitions (**make**, error)
The number of macros you have created exceeds the capacity of your computer to process them.

too many structure initializers (**cc0**, error)
The program contains a structure initialization that has more values than members.

trailing "," in initialization list (**cc0**, warning)
An initialization statement ends with a comma, which is legal.

type clash (**cc0**, error)
The parser expected to find matching types but did not. For example, the types of **e1** and **e2** in **(x) ? e1 : e2** must either both be pointers or neither be pointers.

type of function "*string*" adjusted to *string* (**cc0**, warning)
This warning is given when the type of a numeric constant is widened to **unsigned**, **long**, or **unsigned long** to preserve the constant's value. The type of the constant may be explicitly specified with the **u** or **L** constant suffixes.

type of parameter "*string*" adjusted to *string* (**cc0**, warning)
The program uses a parameter that the C language says must be adjusted to a wider type, e.g., **char** to **int** or **float** to **double**.

type required in cast (**cc0**, error)
: The type is missing from a cast declaration.

undefined local symbol *string* (**as**, error)
: The program uses the symbol *string*, but never defines it.

unexpected end of enumeration list (**cc0**, error)
: An end-of-file flag or a right brace occurred in the middle of the list of enumerators.

unexpected end of line (**as**, error)
: The present expression ends abruptly, and may have been truncated.

unexpected EOF (**cc0**, **cc1**, **cc2**, **cc3**, fatal)
: **EOF** occurred in the middle of a statement. The temporary file may have been corrupted or truncated accidentally. Check your disk drive to see that it is working correctly. Also, make sure that you did not accidentally embed a **<ctrl-Z>** in the line.

union "*string*" does not contain member *m* (**cc0**, error)
: The program attempted to address the variable string *m*, which is not defined as part of the structure *string*.

unknown operator (**as**, error)
: The program used an operator that **as** does not recognize.

string: unknown option (**cpp**, fatal)
: The preprocessor **cpp** does not recognize the option *string*. Try re-typing the **cc** command line.

= without macro name or in token list (**make**, error)
: An equal sign '**=**' can be used only to define a macro, using the following syntax: "MACRO=*definition*". An incomplete macro definition, or the appearance of an equal sign outside the context of a macro definition, will trigger this error message.

: without preceding target (**make**, error)
: A colon appeared without a target file name, e.g., :*string*.

write error on output object file (**cc2**, fatal)
: **cc2** could not write the relocatable object module. Most likely, your mass storage device has run out of room. Check to see that your disk drive or hard disk has enough room to hold the object module, and that it is working correctly.

zero modulus (**cc0**, warning)
: The program will perform a modulo operation by zero if the code just parsed is executed. Although the program can be parsed, this statement may create trouble if executed.

The Lexicon

The rest of this manual consists of the Lexicon. The Lexicon consists of several hundred articles, each of which describes a function or command, defines a term, or otherwise gives you useful information. The articles are organized in alphabetical order.

Internally, the Lexicon has a *tree structure*. The "root" entry is the one for **Lexicon**. It, in turn, refers to a series of **Overview** entries. Each Overview entry introduces a group of entries; for example, the Overview entry for **string** introduces all of the string functions and macros, lists them, and gives a lengthy example of how to use them.

Each entry cross-references other entries. These cross-references point up the documentation tree, to an overview article and, ultimately, to the entry for **Lexicon** itself; down the tree to subordinate entries; and across to entries on related subjects. For example, the entry for **getchar** cross-references **STDIO**, which is its Overview article, plus **putchar** and **getc**, which are related entries of interest to the user. The Lexicon is designed so that you can trace from any one entry to any other, simply by following the chain of cross-references up and down the documentation tree. Other entries refer to *The Art of Computer Programming* and the first edition of *The C Programming Language*.

For more information on how to use the Lexicon and how it is organized, see the entry in the Lexicon on **Lexicon**.

example — Example

Give an example of Mark Williams Lexicon format
#include <example.h>
char ***example(int** *foo*, **long** *bar*);**

This is an example of the Mark Williams Lexicon format of software documentation. At this point, each entry has a brief narration that discusses the topic in detail.

The lines in **boldface** describe how to use the function being described. The first line, **#include <example.h>**, indicates that this function requires the imaginary header file **example.h**. The second line gives the syntax of the function. **char *example** means that the imaginary function **example** returns a pointer to a **char**. *foo* and *bar* are **example**'s arguments: *foo* must be declared to be an **int**, and *bar* must be declared to be a **long**.

Example

The following program gives an example of an example.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      printf("Many entries include examples\n");
      return EXIT_SUCCESS;
}
```

Cross-references

Standard, §a reference to the ANSI Standard
The C Programming Language, ed. 2, page number

See Also

Lexicon

Notes

If a Lexicon entry uses a technical term that you do not understand, look it up in the Lexicon. In this way, you will gain a secure understanding of how to use **Let's C**.

! to ~

! — Operator

Logical negation operator
!*operand*

The operator **!** is the logical negation operator. Its operand must be an expression with scalar type. **!** then inverts the logical result of its operand. This result has type **int**.

If *operand* is nonzero, **!***operand* yields zero; if *operand* is zero, then **!***operand* yields one.

The expression **!***operand* is equivalent to **(0==***operand***)**.

Cross-references

Standard, §3.3.3.3
The C Programming Language, ed. 2, p. 204

See Also

!=, ~, expressions

!= — Operator

Inequality operator
operand1 **!=** *operand2*

The operator **!=** compares *operand1* with *operand2*. The result of this operation is one if the operands are *not* equal, and zero if they are.

The operands must be one of the following:

- Arithmetic types.
- Pointers to compatible types (ignoring qualifiers on these types).
- A pointer to an object or incomplete type, and a pointer to **void**.
- A pointer and NULL.

If both operands have arithmetic type, they undergo usual arithmetic conversion before being compared. If one operand is a pointer to an object and the other is a pointer to **void**, the pointer to an object is converted to a pointer to **void** for purposes of the comparison.

Cross-references

Standard, §3.3.9
The C Programming Language, ed. 2, pp. 41, 207

See Also

!, ==, expressions

” — Punctuator

String literal character

The quotation mark “” marks the beginning and end of a string literal. To embed a quotation mark within a string literal, use the escape sequence \".

Cross-references

Standard, §3.1.2.5
The C Programming Language, ed. 2, p. 194

See Also

string literal

—

String-ize operator

The operator **#** is read and translated by the preprocessor. It must be followed by one of the formal parameters of a function-like macro. The token sequence that would have replaced the formal parameter in the absence of the **#** is instead converted to a string literal, and the string literal replaces the both the **#** and the formal parameter. This process is called *string-izing*.

For example, the consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the following line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

The preprocessor replaced **#x** with a string literal that names the sequence of token that replaces **x**.

The following rules apply to interpreting the **#** operator:

1. If a sequence of white-space characters occurs within the preprocessing tokens that replace the argument, it is replaced with one space character.
2. All white-space characters that occur before the first preprocessing token and after the last preprocessing token is deleted.
3. The original spelling of the token that is stringized is retained in the string produced. This means that as the string is formed, the translator appropriately escapes any backslashes or quotation marks in the tokens.

Example

The following uses the operator **#** to display the result of several mathematics routines.

```
#include <errno.h>
#include <math.h>
#include <stddef.h>
#include <stdio.h>

void show(double value, char *name)
{
      if (errno)
            perror(name);
      else
            printf("%10g %s\n", value, name);
      errno = 0;
}

#define display(x) show((double)(x), #x)

main(void)
{
      extern char *gets();
      double x;
      char string[64];
```

```
      for(;;) {
            printf("Enter a number: ");
            fflush(stdout);
            if(gets(string) == NULL)
                  break;

            x = atof(string);
            display(x);
            display(cos(x));
            display(sin(x));
            display(tan(x));
            display(acos(cos(x)));
      }
}
```

Cross-references

Standard, §3.8.3.2
The C Programming Language, ed. 2, pp. 90, 230

See Also

##, #define, preprocessing

— Operator

Token-pasting operator

The operator **##** is is used by the preprocessor. It can be used in both object-like and function-like macros. When used immediately before or immediately after an element in the macro's replacement list, it joins the corresponding preprocessor token with its neighbor. This is sometimes called "token pasting".

As an example of token pasting, consider the macro:

```
#define printvar(number) printf("%s\n", variable ## number)
```

When the preprocessor reads the following line

```
printvar(5);
```

it substitutes the following code for it:

```
printf("%s\n", variable5);
```

The preprocessor throws away all white space both before and after the **##** operator.

The **##** operator must not be used as the first or last entry in a replacement list.

All instances of the **##** operator are resolved before further macro replacement is performed.

Cross-references

Standard, §3.8.3.3
The C Programming Language, ed. 2, pp. 90, 230

See Also

#, #define, preprocessing

Notes

Some pre-ANSI translators supported token pasting by replacing a comment in a macro replacement list with no space. ANSI translators always replace a comment with one space, no matter where that comment appears.

LEXICON

The order of evaluation of multiple **##** operators is unspecified.

#define — Preprocessing directive

Define an identifier as a macro
#define *identifier replacement-list*
#define *identifier* **(** *parameter-list*$_{opt}$ **)** *replacement-list*

The preprocessing directive **#define** tells the preprocessor to regard *identifier* as a macro.

#define can define two kinds of macros: *object-like*, and *function-like*.

Object-like Macros

An object-like macro has the syntax

```
#define identifier replacement-list
```

This type of macro is also called a *manifest constant*.

The preprocessor searches for *identifier* throughout the text of the translation unit, excluding comments, string literals, and character constants, and replaces it with the elements of *replacement-list*, which is then rescanned for further macro substitutions.

For example, consider the directive:

```
#define BUFFERSIZE 75
```

When the preprocessor reads the line

```
malloc(BUFFERSIZE);
```

it replaces it with:

```
malloc(75);
```

Function-like Macros

A function-like macro is more complex. The preprocessor looks for *identifier***(***argument-list***)** throughout the text of the translation unit, excluding comments, string literals, and character constants. The number of comma-separated arguments in *argument-list* must match the number of comma-separated parameters in the *parameter-list* of the macro's definition. The list is optional in the sense that some function-like macros do not have any parameters.

In the following description, *argument* means the sequence of tokens in *argument-list* that occupies the same relative position as the parameter under discussion occupies in *parameter-list*. The preprocessor replaces *identifier***(***argument-list***)** with the *replacement-list* specified in the definition after it performs the following substitutions: If a parameter is followed or preceded by the operator **##**, then the parameter is replaced by the argument. If a parameter is preceded by **#**, then the **#** and the parameter are replaced by a string literal that contains the argument. All other instances of parameters are replaced by the argument after the argument has first been exhaustively scanned for further preprocessor macro expansions. All instances of **##** are converted to token-paste operations.

For example, the consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the following line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

When an argument to a function-like macro contains no preprocessing tokens, or when an argument to a function-like macro contains a preprocessing token that is identical to a preprocessing directive, the behavior is undefined.

Macro Rescanning

As noted above, the preprocessor searches for macro identifiers throughout the text of the translation unit, excluding comments, string literals, and character constants. The text of replaced macros is also scanned for macro replacements, but it is not part of the text of the translation unit (i.e., source file), so it does not follow the same rules.

After it replaces the identifier of an object-like macro or the *identifier***(***argument-list***)** of a function-like macro with the appropriate *replacement-list*, the preprocessor continues to scan for further macro invocations, starting with the *replacement-list*.

While the preprocessor scans the *replacement-list*, it suppresses the definition of the macro that produced the list. If the preprocessor recognizes a second macro invocation and replaces it before it processes the tokens that replace the first invocation, then it suppresses the definitions of both the first and the second macros while it processes the *replacement-list* of the second macro.

The preprocessor suppresses a definition as long as any of the tokens that remain to be processed are derived directly from the original macro replacement or from further macro replacements that use parts of the original macro replacement. Thus, when the object-like macro definition

```
#define RECURSE     RE ## CURSE
```

is invoked by the token **RECURSE**, it is replaced by the token **RECURSE** formed by pasting **RE** and **CURSE** together, but the scanning of the replacement list would not invoke the macro RECURSE a second time. Likewise, the function-like macro definition

```
#define RECURSE(a, b)   a ## b(a, b)
```

when invoked with the sequence **RECURSE(RE, CURSE)** would be replaced by the token sequence **RECURSE(RE, CURSE)**, but the scanning of the replaced token sequence would not invoke the macro **RECURSE()** again.

Be warned that you should not test a PC-based compiler for compliance with these macro definitions unless you are prepared to turn off your machine. If the compiler fails to detect the recursion, it may become locked in an infinite loop, and there may be no other way to terminate the substitution.

Example

For an example of using a function-like macro in a program, see **#**.

Cross-references

Standard, §3.8.3
The C Programming Language, ed. 2, pp. 229*ff*

See Also

#, ##, #undef, preprocessing

Notes

A macro expansion always occupies exactly one line, no matter how many lines are spanned by the definition or the actual parameters.

A macro definition can extend over more than one line, provided that a backslash '\' appears before the newline character that breaks the lines. The size of a **#define** directive is therefore limited by the maximum size of a logical source line, which can be up to at least 509 characters long.

LEXICON

A macro may be redefined only if the new definition matches the old definition in all respects except the spelling of white space.

#elif — Preprocessing directive

Include code conditionally
#elif *constant-expression* <*newline*> *group*$_{opt}$

The preprocessing directive **#elif** conditionally includes code within a program. It can be used after any of the instructions **#if**, **#ifdef**, or **#ifndef**, and before **#endif** that ends the chain of conditional-inclusion directives.

If the conditional expression of the preceding **#if**, **#ifdef**, or **#ifndef** directive is false and the *constant-expression* that follows **#elif** is non-zero, then *group* is included within the program up to the next **#elif**, **#else**, or **#endif** directive. An **#if**, **#ifdef**, or **#ifndef** directive may be followed by any number of **#elif** directives.

The *constant-expression* must be an integral expression, and it cannot include a **sizeof** operator, a cast, or an enumeration constant. All macro substitutions are performed upon the *constant-expression* before it is evaluated. All integer constants are treated as long objects, and are then evaluated. If *constant-expression* includes character constants, all escape sequences are converted into characters before evaluation. The implementation defines whether the result of evaluating a character constant in *constant-expression* matches the result of evaluating the same character constant in a C expression. For example, it is up to the implementation whether

```
#elif 'z' - 'a' == 25
```

yields the same value as:

```
else if ('z' - 'a' == 25)
```

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#else, #endif, #if, #ifdef, #ifndef, preprocessing

#else — Preprocessing directive

Include code conditionally
#else *newline group*$_{opt}$

The preprocessing directive **#else** conditionally includes code within a program. It is preceded by one of the directives **#if**, **#ifdef**, or **#ifndef**, and may also be preceded by any number of **#elif** directives. If all preceding directives evaluate to false, then the code introduced by **#else** is included within the program up to the **#endif** directive that concludes the chain of conditional-inclusion directives.

A **#if**, **#ifdef**, or **#ifndef** directive can be followed by only one **#else** directive.

Example

For an example of using this directive in a program, see **assert**.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #endif, #if, #ifdef, #ifndef, preprocessing

#endif — Preprocessing directive

End conditional inclusion of code
#endif

The preprocessing directive **#endif** must follow any **#if**, **#ifdef**, or **#ifndef** directive. It may also be preceded by any number of **#elif** directives and an **#else** directive. It marks the end of a sequence of source-file statements that are included conditionally by the preprocessor.

Example

For an example of using this directive in a program, see **assert**.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #else, #if, #ifdef, #ifndef, preprocessing

#error — Preprocessing directive

Error directive
#error *message newline*

The preprocessing directive **#error** prints *message* when an error occurs.

Cross-references

Standard, §3.8.5
The C Programming Language, ed. 2, p. 233

See Also

preprocessing

#if — Preprocessing directive

Include code conditionally
#if *constant-expression newline group*$_{opt}$

The preprocessing directive **#if** tells the preprocessor that if *constant-expression* is true, then include the following lines of code within the program until it reads the next **#elif**, **#else**, or **#endif** directive.

The *constant-expression* must be an integral expression, and it cannot include a **sizeof** operator, a cast, or an enumeration constant. All macro substitutions are performed upon the *constant-expression* before it is evaluated. All integer constants are treated as long objects, and are then evaluated. If *constant-expression* includes character constants, all escape sequences are converted into characters before evaluation.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #else, #endif, #ifdef, #ifndef, preprocessing

LEXICON

Notes

The keyword **defined** determines whether a symbol is defined to **#if**. For example,

```
#if defined(SYMBOL)
```

or

```
#if defined SYMBOL
```

is equivalent to

```
#ifdef SYMBOL
```

except that it can be used in more complex expressions, such as

```
#if defined FOO && defined BAR && FOO==10
```

#ifdef — Preprocessing directive

Include code conditionally
#ifdef *identifier newline group*$_{opt}$

The preprocessing directive **#ifdef** checks whether *identifier* has been defined as a macro or manifest constant. If *identifier* has been defined, then the preprocessor includes *group* within the program, up to the next **#elif**, **#else**, or **#endif** directive. If *identifier* has not been defined, however, then *group* is skipped.

An **#ifdef** directive can be followed by any number of **#elif** directives, by one **#else** directive, and must be followed by an **#endif** directive.

Example

For an example of using this directive in a program, see **assert**.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #else, #endif, #if, #ifndef, defined, preprocessing

Notes

This is the same as:

```
#if defined IDENTIFIER
```

#ifndef — Preprocessing directive

Include code conditionally
#ifndef *identifier newline group*$_{opt}$

The preprocessing directive **#ifndef** checks whether *identifier* has been defined as a macro or manifest constant. If *identifier* has *not* been defined, then the preprocessor includes *group* within the program up to the next **#elif**, **#else**, or **#endif** directive. If *identifier* has been defined, however, then *group* is skipped.

An **#ifndef** directive can be followed by any number of **#elif** directives, by one **#else** directive, and by one **#elif** directive.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #else, #endif, #if, #ifndef, defined, preprocessing

Notes

This is the same as:

```
#if !defined IDENTIFIER
```

#include — Preprocessing directive

Read another file and include it
#include <*file***>**
#include "*file***"**

The preprocessing directive **#include** tells the preprocessor to replace the directive with the contents of *file*.

The directive can take one of two forms: either the name of the file is enclosed within angle brackets (**<***file***>**), or it is enclosed within quotation marks ("*file*"). The name of the file can be enclosed within angle brackets (**<***file***.h>**) or quotation marks ("*file***.h**"). Angle brackets tell the preprocessor to look for *file* in the directories named with the **-I** options to the **cc** command line, and then in the directory named by the environmental variable **INCDIR**. Quotation marks tell **cpp** to look for *file***.h** in the source file's directory, then in directories named with the **-I** options, and then in the directory named by the environmental variable **INCDIR**. **#include** directives may be nested up to at least eight deep. That is to say, a file included by an **#include** directive may use an **#include** directive to include a third file. That third file may also use a **#include** directive to include a fourth file, and so on, up to at least eight files.

A subordinate header is sought relative to the original source file, rather than relative to the header that calls it directly. For example, suppose that under the UNIX operating system, a file **example.c** resides in directory **/v/fred/src**. If **example.c** contains the directive **#include <header1.h>**. The operating system will look for **header1.h** in the standard directory, **/usr/include**. If **header1.h** includes the directive **#include <../header2.h>** then the implementation should look for **header2.h** not in directory **/usr**, but in directory **/v/fred/src**.

Some file systems allow characters to be used in file names that are used as delimiters in other file systems. Therefore, if any of the characters '*', "\', or ',' are part of a file name, behavior is undefined. If "" is part of a file name between angle-bracket delimiters, behavior is also undefined.

A **#include** directive may also take the form **#include** *string*, where *string* is a macro that expands into either of the two forms described above.

Cross-references

Standard, §2.2.4.1, §3.8.2
The C Programming Language, ed. 2, p. 88

See Also

header, header names, Language, preprocessing

Notes

Trigraphs that occur within a **#include** directive are substituted, because they are processed by an earlier phase of translation than are **#include** directives.

LEXICON

#line — Preprocessing directive

Reset line number
#line *number newline*
#line *number filename newline*
#line *macros newline*

#line is a preprocessing directive that resets the line number within a file. The Standard defines the line number as being the number of newline characters read, plus one.

#line can take any of three forms. The first, **#line** *number*, resets the current line number in the source file to *number*. The second, **#line** *number filename*, resets the line number to *number* and changes the name of the file referred to by _ _**FILE**_ _ to *filename*. The third, **#line** *macros*, contains macros that have been defined by earlier preprocessing directives. When the macros have been expanded by the preprocessor, the **#line** instruction will then resemble one of the first two forms and be interpreted appropriately.

number specifies the number of the next source line in the file, not the number of the **#line** directive's source line.

Cross-references

Standard, §3.8.4
The C Programming Language, ed. 2, p. 233

See Also

preprocessing

Notes

Most often, **#line** is used to ensure that error messages point to the correct line in the program's source code. A program generator may use this directive to associate errors in generated C code with the original sources. For example, the program generator **yacc** uses **#line** instructions to link the C code it generates with the **yacc** code written by the programmer.

#pragma — Preprocessing directive

Perform implementation-defined task
#pragma *preprocessing-tokens*$_{opt}$ *newline*

The preprocessing directive **#pragma** causes the implementation to behave in an implementation-defined manner. A **#pragma** might be used to give a "hint" to the translator about the best way to generate code, optimize, or diagnose errors. It may also pass information to the translator about the environment, or add debugging information. The design of **#pragma** is left up to the implementation.

Cross-references

Standard, §3.8.6
The C Programming Language, ed. 2, p. 233

See Also

preprocessing

Notes

An unrecognized pragma is ignored. Because of this subtlety, one should be careful when porting code that contains pragmas to other implementations.

As of this writing, no Mark Williams compiler uses **#pragma**.

#undef — Preprocessing directive

Undefine a macro
#undef *identifier*

The preprocessing directive **#undef** tells the C preprocessor to disregard *identifier* as a manifest constant or macro. It undoes the effect of the **#define** directive.

#undef does not give an error if *identifier* is not defined. It can also undefine macros that are predefined by the implementation, other than those specified by the Standard to be unreadable.

Cross-references

Standard, §3.8.3.5
The C Programming Language, ed. 2, p. 230

See Also

#define, preprocessing

Notes

If an implementation has defined a function both as a macro and as a library function, then the directive

```
#undef function
```

undefines the macro version, and forces the implementation to use the library version.

Some previous implementations allowed a user to "stack" macro definitions and "unstack" them by **#undef**ing them one level at a time. The Standard, however, states that one **#undef** directive undefines all previous definitions.

% — Operator

Remainder operator
operand1 **%** *operand2*

The operator **%** divides *operand1* by *operand2* and yields the remainder.

Both *operand1* and *operand2* must have integral type. Both undergo the usual arithmetic conversions before they are divided, and the type of the result is that to which the operands were converted. If *operand2* is zero, the behavior is undefined. If either operand is negative, the sign of the result is implementation-defined.

The remainder operation normally throws away the quotient. The division operator **/** returns the quotient of a division operation, and throws away the remainder. If you wish to obtain both quotient and remainder, use the functions **div** or **ldiv**. To obtain the remainder from floating-point division, use the function **fmod**.

Cross-references

Standard, §3.3.5
The C Programming Language, ed. 2, p. 205

See Also

%= — Operator

Remainder assignment operator
operand1 **%=** *operand2*

The operator **%=** divides *operand1* by *operand2* and assigns the remainder to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 % operand2
```

Each operand must have an integral type. If the value of *operand2* is zero, the result is undefined.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

%, expressions

& — Operator

&*operand*
operand1 **&** *operand2*
The operator **&** has two meanings, depending upon whether it has one operand or two. In the former instance, it yields the address of its operand. In the latter instance, it performs a bitwise AND operation upon its operands.

Address-of Operator

When used with one operand, **&** yields the value of the address of its operand in the form of a pointer to the type of its operand. The operand must be an lvalue or function designator, with the following restrictions: the operand may not be a bitfield, and it may not be declared with the storage-class specifier **register**. The resulting pointer has the type "pointer to *type*", where *type* is the type of the operand.

ANSI C allows you to take the address of a function or array.

Bitwise AND Operator

When used with two operands, **&** performs a bitwise AND operation. Each operand must have integral type. Each undergoes the normal arithmetic conversions before the operation. **&** yields a result whose type is the same as the promoted operands.

A bitwise AND operation compares the operands bit by bit. It sets a bit in the object it creates only if the corresponding bits in both operands are set.

For example, consider an environment that uses extended ASCII. Here, the character ')' has the bit pattern:

```
0010 1001
```

and the character 'L' has the bit pattern:

```
0100 1100
```

The operation **')'&'L'** yields an object with the following bit pattern:

```
0000 1000
```

Only one bit was set in the result because in only one instance were both corresponding bits set in the operands.

The **&** operation is sometimes called the "intersection" of two bit sets.

Cross-references

Standard, §3.3.3.2, §3.3.10
The C Programming Language, ed. 2, pp. 48, 93

See Also

expressions

&& — Operator

Logical AND operator
operand1 **&&** *operand2*

The operator **&&** performs a logical AND operation. Both *operand1* and *operand2* must have scalar type.

The result of this operation has type **int**. The result has a value of one if both operands are true (i.e., nonzero). If either operand is false (zero), then the result has a value of zero.

The operands are evaluated from left to right. If *operand1* is false, then *operand2* is not evaluated. If *operand2* is an expression that yields a side-effect, the results of the **&&** operation may not be what you expect. If *operand1* is false, *operand2* is not evaluated and its side-effect not generated.

Cross-references

Standard, §3.3.13
The C Programming Language, ed. 2, p. 207

See Also

||, expressions

&= — Operator

Bitwise-AND assignment operator
operand1 **&=** *operand2*

The operator **&=** performs a bitwise AND operation on *operand1* and *operand2* and assigns the result to *operand1*. It is equivalent to the expression

```
operand1 = operand1 & operand2
```

Both operands must have integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

&, expressions

() — Punctuator

functionname **(** *arguments* **)**
(*newtype* **)** *identifier*
(*primary expression* **)**
The characters **()** have two uses in the C world: as punctuators and as operators. Parentheses must be used in pairs.

When the parentheses follow an identifier, they indicate that it names a function. When used with a

function declaration, a function prototype, or a function definition, the parentheses may enclose a list of parameters for the function and the type of each parameter. When used with a function call, they enclose a list of arguments to be passed to the function.

When parentheses precede an identifier and enclose a typename alone, then they function as the cast operator. Here, the type of the identifier is changed, or *cast*, to the type enclosed within parentheses.

Finally, when parentheses enclose an expression, that expression is by definition considered to be a primary expression. This means that the expression is resolved before any outer expression is evaluated.

To see the variety of uses for **()**, consider the following expression:

```
if ((fileptr = (void *)fopen("filename", "r")) == NULL)
```

The outermost pair of parentheses enclose the arguments to **if**. The next innermost pair of parentheses enclose the expression

```
fileptr = (void *)fopen("filename", "r")
```

which must be resolved before it is compared with NULL. The pair of parentheses that enclose the type **void** * casts the object returned by **fopen** to type **void** *. Finally, the parentheses that follow **fopen** mark that identifier as a function and enclose the arguments that are passed to it, in this case the string literals **filename** and **r**.

Cross-reference

Standard, §3.1.6, §3.3.2.2, §3.3.4

See Also

function calls, function definition, function prototype, operators, punctuators

Notes

Under ANSI C, parentheses affect the grouping of expressions. This is a quiet change from the definition in the first edition of *The C Programming Language*, which allowed translators to rearrange expressions in the presence of parentheses on expressions that involved commutative and associative operators (binary **+** and *). The *as if* rule still applies in this case: if the translator can produce the same results, it is free to rearrange expressions in the face of parentheses.

* — Operator

**pointer*
*typename * type-qualifier-list*$_{opt}$ *identifier*
*operand1 * operand2*
The character * is used both as an operator and as a punctuator.

Multiplication Operator

When the * appears between two operands with arithmetic type, it is the multiplicative operator. It multiplies its operands and yields the product. Both operands undergo normal arithmetic conversion. The type of the result is the one to which both operands were converted.

Indirection Operator

When * is used before one operand that is of a pointer type, it *dereferences* the pointer. That is, it yields the value of the object to which the pointer points. If the pointer points to a function, then the result is a function designator. If the pointer points to an object, the resulting lvalue has the type of the object to which the pointer points.

If indirection is performed on any pointer to an incomplete type, the behavior is undefined. This means that no pointer with type **void** * can be dereferenced.

Pointer Punctuator

When the * is used in a declaration, it indicates that the variable being declared is a pointer. For example, consider the following:

```
int example1;
int *example2;
```

Here, **example1** has type **int**, and **example2** has type "pointer to **int**".

Cross-references

Standard, §3.1.6, .3.3.2, §3.3.5, §3.5.4.1
The C Programming Language, ed. 2, pp. 94, 205

See Also

expressions, operators, pointer, punctuators

*/ — Comment delimiter

The characters ***/** together mark the end of a comment.

Cross-references

Standard, §3.1.9
The C Programming Language, ed. 2, p. 192

See Also

/*, comment

*= — Operator

Multiplication assignment operator
operand1 ***=*** *operand2*

The operator ***=*** multiplies *operand1* by *operand2* and assigns the product to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 * operand2
```

Each operand must have an arithmetic type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

***, expressions**

+ — Operator

+*operand*
operand1 **+** *operand2*
The operator **+** has two uses, depending upon whether it is used with two operands or one. In the former instance, it indicates that the given operand should be computed without any associative or commutative regrouping that the translator might normally apply to expressions. In the latter, it adds the two operands together.

The Unary + Operator

The unary operator **+** takes an operand that has a scalar type and yields its value. If the operand has a negative value, then a negative value is returned. The operand undergoes integral promotion,

and the type returned is that to which the operand is promoted.

The Addition Operator

The addition operator **+** adds two operands. Both operands may have arithmetic types, or one of the operands may be a pointer and the other an integral type.

If both operands have arithmetic types, then each undergoes integral conversion before addition is performed; the type of the result is the type to which both are converted.

When an integral type is added to a pointer, the value of the integral operand is first multiplied by the size of the object to which the pointer points, in bytes, and then addition is performed. The result of the addition operation returns a pointer that is appropriately offset from the pointer operand.

Pointer addition is often used for pointers that point to arrays. Note the following rules for incrementing a pointer to an array:

- If a pointer points to an array, then the result of addition will point to another member of the same array — assuming that the array is large enough.
- If a pointer to an array is incremented and the resulting pointer does *not* point to a member of the array or one past the last member, then behavior is undefined.
- Behavior is also undefined if the pointer operand and the result of the addition operation do not point to the same array object *and* the result of the addition operation is then redirected with the unary * operator. In other words, it is legal for a translator to test array bounds.

Cross-references

Standard, §3.3.3.3, §3.3.6
The C Programming Language, ed. 2, pp. 203, 205

See Also

++, -, expressions

++ — Operator

Increment operator
operand++
++*operand*

The operator **++** increments its operand. When it appears before its operand, it is called the *pre-increment operator*; when it appears after its operand, it is called the *post-increment operator*. In both cases, it is equivalent to *operand* **=** *operand***+1**. *operand* must be a modifiable lvalue.

These operators differ as follows: with the prefix operator, the value of the operand is used *after* it is incremented; whereas with the postfix operator, the value of the operand is used *before* it is incremented.

The following example illustrates the difference between the preincrement and postincrement operators.

```
#define MAX  10
int x = 0, count = 0;

/* loop 1 */
while (++x < MAX)
      count++;
```

```
/* loop 2 */
while (x++ < MAX)
      count++;
```

The first loop will iterate nine times, the second will iterate ten times. The first loop preincrements the loop variable **x** before using it within the conditional expression. The second loop, which uses the postincrement operator, first uses the current value of **x** in the conditional, then increments its value.

Cross-references

Standard, §3.3.2.4, §3.3.3.1
The C Programming Language, ed. 2, p. 46

See Also

--, expressions

+= — Operator

Addition assignment operator
operand1 **+=** *operand2*

The operator **+=** adds the value of *operand1* with that of *operand2* and stores the sum within *operand1*. It is equivalent to the expression:

```
operand1 = operand1 + operand2
```

Both operands have arithmetic types, or *operand1* has a pointer type and *operand2* has integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

-=, expressions

Notes

The lvalue *operand1* is evaluated only once.

, — Operator

identifier1 **,** *identifier2*
expression1 **,** *expression2*
The character ',' can be used as punctuator or an operator.

The Comma Punctuator

When it is used as a punctuator, the comma separates the parameters in a function declaration, the parameters to a function-like macro, the arguments to a function call, or the items in a list of identifiers. For example, in the expression

```
int foo, bar, baz;
```

the comma separates the identifiers being declared, all of which are of type **int**.

The Comma Operator

When used outside of a declaration or parameter list, the comma acts as an operator. The comma operator evaluates its left argument first, then its right argument. The value and type of the comma expression is that of the right operand.

LEXICON

For example, the following shows how the comma operator is used in a loop:

```
int i, j;
   . . .
for (i=j=0; i<10 && j<25; i++, j++);
```

This loop uses the comma operator to help increment two variables upon each iteration.

Cross-references

Standard, §3.3.17
The C Programming Language, ed. 2, p. 62

See Also

expressions

Notes

A comma expression cannot be an lvalue.

- — Operator

-operand
operand1 - operand2
The operator **-** has two uses, depending upon whether it is used with two operands or one. In the former situation, it subtracts the operand to its right from the operand to its left. In the latter, it returns the negated value of its operand.

Subtraction Operator

The operator **-** can subtract the following operands from each other:

- Two arithmetic types.
- Two pointers to objects that have compatible types and compatible qualification.
- Two pointers that point to objects that have compatible types, but not necessarily compatible qualification.
- An integral type from a pointer.

When both operands have arithmetic type, each undergoes integral promotion. The type of the result is that to which the operands were promoted. Its value is the difference when the right operand is subtracted from the left.

When one pointer is subtracted from another, the result is of type **ptrdiff_t**. This type is defined in the header **stddef.h**. If two pointers that do not point to the same array are subtracted from each other, behavior is undefined. The only exception is the expression

```
(X+1) - X
```

where, if **X** points to the last member of the array, the result is one by definition.

If two pointers that point to the same array are subtracted from each other, the result is automatically divided by the size of an array member. This yields a value that is the same as would result if the two appropriate array subscripts had been subtracted from each other. If the result of pointer subtraction points past the end of an array, the behavior is undefined. The sole exception, again, is the expression given above.

When subtracting a scalar from a pointer, the result is as if the scalar were multiplied by the size of the object pointed to by the pointer, and then subtracted.

The unary operator **-** takes an operand with arithmetic type. The operand first undergoes normal integral promotion. The type of the resulting expression is the one to which the operand was promoted; and the value of the resulting expression is the negated value of the operand.

Cross-references

Standard, §3.3.3.3, §3.3.6
The C Programming Language, ed. 2, pp. 203, 205

See Also

+, --, expressions

-- — Operator

Decrement operator
*operand***--**
--*operand*

The operator **--** decrements its operand. When it appears before its operand, it is called the *pre-decrement operator*; when it appears after its operand, it is called the *post-decrement operator*. In both cases, it is equivalent to *operand* **=** *operand* **- 1**.

These operators differ as follows: with the prefix operator, the value of the operand is used *after* it is decremented; whereas with the postfix operator, the value of the operand is used *before* it is decremented.

Cross-references

Standard, §3.3.2.4, §3.3.3.1
The C Programming Language, ed. 2, p. 46

See Also

++, expressions

-= — Operator

Subtraction assignment operator
operand1 **-=** *operand2*

The operator **-=** subtracts the value of *operand2* from that of *operand1* and stores the difference within *operand1*. It is equivalent to the expression:

```
operand1 = operand1 - operand2
```

Both operands have arithmetic types, or *operand1* has pointer type and *operand2* has integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

+=, expressions

-> — Operator

Select a member
objectpointer **->** *membername*

The operator **->** selects a member of a structure or a **union** through a pointer.

objectpointer must point to a structure or **union**. *membername* must name a member of the structure or **union** to which *objectpointer* points. For example, consider the following:

```
struct example {
      int member1;
      long member2;
      example *member3;
};

struct example structure;
struct example *pointer = &structure;
```

To select **member1** within **structure** via **pointer**, use the expression:

```
pointer->member1
```

Behavior is implementation-defined if one member of a **union** is accessed after another member has been stored within the **union**.

Cross-references

Standard, §3.3.2.3
The C Programming Language, ed. 2, p. 131

See Also

., expressions, operators

. — Operator

Member selection
objectname **.** *membername*

The operator **.** is used to select a member of a structure or a **union**.

objectname must name a structure or **union**. *membername* must be a member of the structure or **union** that *objectname* names. For example, consider the following:

```
struct example {
      int member1;
      long member2;
      example *member3;
};

struct example object;
```

To read **member1** within **object**, use the expression:

```
object.member1
```

Cross-references

Standard, §3.3.2.3
The C Programming Language, ed. 2, p. 128

See Also

->, expressions, member

/ — Operator

Division operator
operand1 **/** *operand2*

The operator **/** divides *operand2* by *operand1* and yields the quotient. Each operand must have arithmetic type and undergoes the usual arithmetic promotion before the operation is performed. The result of the operation has the type to which the operands are promoted. If the result of **X/Y** can be represented, then **(X/Y)*Y+(X%Y)** must equal **X**.

If *operand2* is zero, the result is undefined. If either operand is negative, the result is either the largest integer that is less than the algebraic quotient, or the smallest integer that is greater than the algebraic quotient, whichever the implementation prefers. For example, in the expression

```
7 / -2
```

the algebraic quotient is **-3.5**. The implementation determines whether the result is **-4** (the largest integer less than the algebraic quotient) or **-3** (the smallest integer greater than the algebraic quotient).

The division operation normally throws away the remainder. The remainder operator **%** returns the remainder of a division operation and throws away the quotient. If you wish to obtain both quotient and remainder, use the functions **div** or **ldiv**.

Cross-references

Standard, §3.3.5
The C Programming Language, ed. 2, p. 205

See Also

%, div, expressions, ldiv

/* — Comment delimiter

The characters **/*** together mark the beginning of a comment.

Cross-references

Standard, §3.1.9
The C Programming Language, ed. 2, p. 192

See Also

***/, comment**

/= — Operator

Division assignment operator
operand1 **/=** *operand2*

The operator **/=** divides *operand1* by *operand2*, and assigns the quotient to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 / operand2
```

Each operand must have arithmetic type.

If the value of *operand2* is zero, behavior is undefined.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

/, expressions

: — Punctuator

When punctuator **:** follows an identifier, it marks the identifier as being a label. When it precedes an integer constant in the declaration of a structure or **union**, it marks the constant as giving the size of a bit-field.

Cross-reference

Standard, §3.1.6
The C Programming Language, ed. 2, p. 66

See Also

?:, bit-fields, goto, label, punctuators

; — Punctuator

The punctuator **;** marks the end of a statement.

Cross-reference

Standard, §3.1.6

See Also

punctuators, statements

< — Operator

Less-than operator
operand1 **<** *operand2*

The operator **<** compares two operands. It yields one if *operand1* is less than *operand2*, and zero if *operand1* is greater than or equal to *operand2*.

See **operators** for more information on the types of operands that can be compared.

Cross-references

Standard, §3.3.8
The C Programming Language, ed. 2, pp. 41, 206

See Also

<=, >, expressions

<< — Operator

Bitwise left-shift operator
operand1 **<<** *operand2*

The operator **<<** shifts the bits in *operand1* to the left by *operand2* places. This is called the *bitwise left shift* operation.

Both operands must have integral types. Both undergo the usual arithmetic conversions, and the result has the type to which the left operand was promoted.

A bitwise left-shift operation moves the bits of an object to the left, and fills the vacated bits with zeroes. For example, consider an environment that uses extended ASCII. Here, the character constant **'?'** has the bit pattern:

```
0011 1111
```

In this environment, the expression

```
'?' << 4
```

yields the following pattern of bits:

```
0000 0011 1111 0000
```

The "nybbles" to the left result from the promotion of the **char** to type **int**. All bits are shifted four places to the left, and the four vacated bits to the right are filled with zeroes.

The left-shift operation is sometimes called the "logical" shift operation, which will fill vacated bits with zeroes.

If *operand2* is negative or is larger than the number of bits in *operand1*, behavior is undefined.

Example

For a practical example of the operator **<<**, see **rand()**.

Cross-references

Standard, §3.3.7
The C Programming Language, ed. 2, pp. 48, 207

See Also

<<=, >>, expressions

<<= — Operator

Bitwise left-shift assignment operator
operand1 **<<=** *operand2*

The operator **<<=** shifts the bits in *operand1* to the left by *operand2* places, and assigns the result to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 << operand2
```

Both operands must have integral type.

If *operand2* is negative or has a value greater than the number of bits in *operand1*, behavior is undefined.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

<<, expressions

<= — Operator

Less-than or equal-to operator
operand1 **<=** *operand2*

The operator **<=** compares two operands. It returns one if *operand1* is less than or equal to *operand2*, and it returns zero if *operand1* is greater than *operand2*.

See **operators** for more information on the types of operands that can be compared.

Example

For an example of using this operator in a program, see **bitwise operators**.

LEXICON

Cross-references

Standard, §3.3.8
The C Programming Language, ed. 2, pp. 41, 206

See Also

<, >=, expressions

= — Operator

Assignment operator
operand1 **=** *operand2*

The operator **=** copies the value of *operand2* into *operand1*. The value of *operand2* is converted to the type of *operand1* before they are copied.

The following types of operands are allowed:

- Both have an arithmetic type. *operand1* may be qualified.
- Both are compatible structures or **union**s. *operand1* may be qualified.
- Both are pointers to compatible types. *operand1* may be a pointer to a qualified type. *operand2* may be NULL. Either may be of type **void** *, assuming the other points to an object or an incomplete type.

operand1 must be a modifiable lvalue.

Cross-references

Standard, §3.3.16.1
The C Programming Language, ed. 2, pp. 50, 208

See Also

==, expressions

== — Operator

Equality operator
operand1 **==** *operand2*

The operator **==** compares *operand1* with *operand2*. The result is one if the operands are equal, and zero if they are not.

The operands must be one of the following:

- Arithmetic types.
- Pointers to compatible types (ignoring qualifiers on these types).
- A pointer to an object or incomplete type and a pointer to **void**.
- A pointer and NULL.

If both operands have arithmetic type, they undergo usual arithmetic conversion before being compared. If one operand is a pointer to an object and the other is a pointer to **void**, the pointer to an object is converted to a pointer to **void** for purposes of the comparison.

If two pointers to functions compare equal, then they point to the same function; likewise, if two pointers to data objects compare equal, then they point to the same object. However, on machines that provide separate spaces for instructions and data, a pointer to a function may compare equal to a pointer to a data object. Therefore, you should not depend on being able to distinguish function pointers from data object pointers by value. Further, on machines that allow many pointer values

to refer to the same object (e.g., i8086 LARGE model), two pointers that do not compare equal may nonetheless point to the same object.

Cross-references

Standard, §3.3.9
The C Programming Language, ed. 2, pp. 41, 207

See Also

!=, expressions

Notes

Perhaps the commonest mistake made by C programmers is to use the assignment operator '=' in place of the equality operator '==' where a conditional expression is expected. For example:

```
if (variable1 = variable2)    /* WRONG */
      dosomething();
```

Here, the value of **variable2** is copied into **variable1**; whether the expression succeeds or not depends upon the value of **variable2** rather than the equality of the two variables. Hence, the condition will be true as long as this operand has a value other than zero. This code will translate, often without generating a warning message, but probably will not run correctly.

Type conversion will affect comparison, particularly if a **char** is being compared with an integral type with a negative value. For example, consider the comparison:

```
char variable;
   . . .
if (variable == -1)
      dosomething();
```

Here, **variable** is promoted to an **int** before it is compared with **-1**. However, if **char** is unsigned by default, when it is expanded, it can never compare equal to a negative number. For maximum portability, when using **char**s that may take negative values, declare them as type **int** or type **signed char**. All Mark Williams compilers used signed **char**s by default.

Comparing **float**s and **double**s for equality is usually a mistake, especially as a control expression in a loop. Implementations of floating-point arithmetic are often inexact.

> — Operator

Greater-than operator
operand1 **>** *operand2*

The operator **>** compares two operands. It returns one if *operand1* is greater than *operand2*. It returns zero if *operand1* is less than, or equal to, *operand2*.

Cross-references

Standard, §3.3.8
The C Programming Language, ed. 2, pp. 41, 206

See Also

<, >=, expressions

>= — Operator

Greater-than or equal-to operator
operand1 **>=** *operand2*

LEXICON

The operator **>=** compares two operands. It returns one if *operand1* is greater than, or equal to, *operand2*; it returns zero if *operand1* is less than *operand2*.

Cross-references

Standard, §3.3.8
The C Programming Language, ed. 2, pp. 41, 206

See Also

<=, >, operators

>> — Operator

Bitwise right-shift operator
operand1 **>>** *operand2*

The operator **>>** shifts the bits in *operand1* to the right by *operand2* places. This is called the *bitwise right shift* operation.

Both operands must have integral type. Both undergo the usual arithmetic conversions, and the result has the type to which the left operand was promoted.

A bitwise right-shift operation moves the bits of an object to the right. The vacated bits are filled with zeroes, unless *operand1* is signed and has a negative value. In that case, the vacated bits will propagate the sign bit (i.e., be filled with ones).

For example, consider an environment that uses extended ASCII. Here, the character constant **'?'** has the bit pattern:

```
0011 1111
```

In this environment, the expression

```
'?' >> 4
```

yields the following pattern of bits:

```
0000 0000 0000 0011
```

The two "nybbles" to the right result from the promotion of the **char** to type **int**. All bits are shifted four places to the right, and the four vacated bits to the left are filled with zeroes. The nybble **1111** disappears.

The right-shift operation is sometimes called the "arithmetic" shift operation.

If *operand2* is negative or is larger than the number of bits in *operand1*, behavior is undefined.

Example

For an example of using this operator in a program, see **srand**.

Cross-references

Standard, §3.3.7
The C Programming Language, ed. 2, pp. 48, 207

See Also

<<, >>=, expressions

>>= — Operator

Bitwise right-shift assignment operator
operand1 **>>=** *operand2*

The operator **>>=** shifts the bits in *operand1* to the right by *operand2* places, and assigns the result to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 >> operand2
```

Both operands must have integral type.

If *operand2* is negative or has a value larger than the number of bits in *operand1*, behavior is undefined.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

>>, expressions

?: — Operator

Conditional operator
conditional **?** *expression1* **:** *expression2*

The conditional operator **?:** causes one or the other of two expressions to be executed.

If the *conditional* evaluates to true (nonzero), then *expression1* is evaluated; otherwise, *expression2* is evaluated. The operator as a whole yields the result of whichever expression is executed.

The *logical-OR-expression* must have a scalar type. The conditional operator may take the following types:

- Both are arithmetic types. Each undergoes normal arithmetic conversion, and the result has the type to which they are converted.
- Both have compatible structure or **union** types. They are converted to a common type, and the result has that type.
- Both are **void** types. The result is of type **void**.
- Both are pointers to compatible types, whether qualified or unqualified. The result is a pointer that is qualified by all the qualifiers of both operands.
- One is a pointer and the other NULL. The result is of the pointer's type.
- One points to an object or incomplete type, and the other is type **void** *. Both operands are converted to type **void** * before evaluation, and the result also has that type.

The logical expression can also be a scalar identifier, constant, or function.

Cross-references

Standard, §3.3.15
The C Programming Language, ed. 2, p. 51

See Also

expressions

LEXICON

Notes

The conditional operator does not yield an lvalue. For example:

```
int x, a, b;
(x ? a : b) = 5;  /* WRONG */
```

is incorrect, but

```
int x;
int *ptr1, *ptr2;
*(x ? ptr1 : ptr2) = 5; /* RIGHT */
```

is correct.

[] — Operator

Array subscript operator
*arrayname***[***size***]**

The array-subscript operator **[]** is used in different contexts. It is used to declare an array, with or without the array size. It is used as a subscript operator, and it can also be used when passing an array as an argument. *arrayname* is the name of the array to be accessed; *size* is the number of objects in the array.

The Standard states that one of the items *arrayname* or *size* must be a pointer and the other an integer. To calculate the address of an element within an array, the integer is multiplied by the size of an element of the array, and the product added to value of the pointer. In most C programs, *arrayname* gives the pointer and *size* the integer offset.

The operator **[]** can also be used to select an object within an array; the objects are numbered from zero through *size*-1. For example, if *arrayname* points to an array of **int**s, and if *size* is equal to six, then the expression

```
arrayname[4]
```

is equivalent to:

```
*(arrayname+4)
```

This expressions yields not an address, but the contents of the array at the requested point.

An array can be followed by more than one pair of brackets. Such arrays are called *multidimensional*. To see how such an array works, consider the following multidimensional array:

```
#define DIMENSION1 5
#define DIMENSION2 10
int arrayname[DIMENSION1][DIMENSION2];
```

Here, **dimension1** holds five objects, each of which is the size set by **dimension2**: in this instance, ten **int**s. Thus, the expression

```
arrayname[3][5];
```

is equivalent to writing:

```
*(arrayname+(3*DIMENSION2)+5)
```

An expression of the form

```
arrayname[3];
```

indicates an entire row of the array. This is sometimes called a “slice”.

Cross-references

Standard, §3.3.2.1
The C Programming Language, ed. 2, pp. 97*ff*

See Also

array, expressions

Notes

Given the Standard's description of how an array is accessed, the elements of an array access may be reversed. For example, given the following code,

```
int arrayname[5];
int counter = 3;
```

the expressions

```
arrayname[counter]
```

and

```
counter[arrayname]
```

should yield the same result. Using these expressions interchangeably will result in programs that are very hard to read and maintain.

^ — Operator

Bitwise exclusive OR operator
operand1 **^** *operand2*

The operator **^** performs an bitwise exclusive OR operation.

Each operand must have integral type, and each undergoes the usual arithmetic conversions. The result has integral type.

A bitwise exclusive OR operation compares the bit patterns of the operands, then sets each bit in its result if either, but not both, of the corresponding bits in the operands is set.

For example, consider an environment which uses extended ASCII. In this environment, the character **9** is represented by the bit pattern

```
0011 1001
```

and the character **w** by the bit pattern:

```
0111 0111
```

Thus, the operation:

```
'9' ^ 'w';
```

yields the following bit pattern:

```
0000 0000 0100 1110
```

The extra "nybbles" to the left are created by the promotion of the character constants to type **int**. If the corresponding bits in the operands were both set to one, the bit in the result was set to zero.

Example

For an example of using this operator in a program, see **srand**.

Cross-references

Standard, §3.3.11
The C Programming Language, ed. 2, pp. 48, 207

See Also

^=, |, expressions

^= — Operator

Bitwise exclusive-OR assignment operator
operand1 **^=** *operand2*

The operator **^=** performs a bitwise exclusive-OR operation on *operand1* and *operand2*, and assigns the result to *operand1*. It is equivalent to the expression

```
operand1 = operand1 ^ operand2;
```

Both operands must have integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

^, expressions

_ _DATE_ _ — Manifest constant

Date of translation

_ _**DATE**_ _ is a manifest constant that is defined by the implementation. It represents the date that the source file was translated. It is a string literal of the form

```
"Mmm dd yyyy"
```

where **Mmm** is the same three-letter abbreviation for the month as is used by **asctime**; **dd** is the day of the month, with the first **d** being a space if translation occurs on the first through the ninth day of the month; and **yyyy** is the current year. If the date of translation is not available, then a valid, implementation-defined date must be supplied.

The value of _ _**DATE**_ _ remains constant throughout the processing of the translation unit. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

_ _FILE_ _, _ _LINE_ _, _ _STDC_ _, _ _TIME_ _, preprocessing

__end — External data

extern char * __end;
__end is an external variable that points to the end of your program's data space. It is set by the C runtime startup, and can be incremented by the function **sbrk**.

See Also

Environment, malloc, maxmem, sbrk

_ _FILE_ _ — Manifest constant

Source file name

_ _FILE_ _ is a manifest constant that is defined by the implementation. It represents, as a string constant, the name of the current source file being translated.

_ _FILE_ _ may not be the subject of a **#define** or **#undef** preprocessing directive, but it may be altered with the **#line** preprocessing directive.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

#line, _ _DATE_ _, _ _LINE_ _, _ _STDC_ _, _ _TIME_ _, preprocessing

_ _LINE_ _ — Manifest constant

Current line within a source file

_ _LINE_ _ is a manifest constant that is defined by the implementation. It represents the current line within the source file. The Standard defines the current line as being the number of newline characters read, plus one.

_ _LINE_ _ may not be the subject of a **#define** or **#undef** preprocessing directive.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

_ _DATE_ _, _ _FILE_ _, _ _STDC_ _, _ _TIME_ _, preprocessing

_ _STDC_ _ — Manifest constant

Mark a conforming translator

_ _STDC_ _ is a manifest constant that is defined by the implementation. If it is defined to be equal to one, then it indicates that the translator conforms to the Standard.

The value of **_ _STDC_ _** remains constant throughout the entire program, no matter how many source files it comprises. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

For an example of using **_ _STDC_ _** in a program, see **assert**.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

_ _DATE_ _, _ _FILE_ _, _ _LINE_ _, _ _TIME_ _, preprocessing

Notes

If an implementation is not fully compatible with the Standard, then it should not define **_ _STDC_ _**. A value greater than one may indicate compliance with a later version of the Standard.

LEXICON

_ _TIME_ _ — Manifest constant

Time source file is translated

_ _TIME_ _ is a manifest constant that is defined by **Let's C**. It represents the time that a source file is translated. It is a string literal of the form:

```
"hh:mm:ss"
```

This is the same format used by the function **asctime**. If the time of translation is not available, then a valid, implementation-defined string must be supplied.

The value of this remains constant throughout the processing of the translation unit. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

_ _DATE_ _, _ _FILE_ _, _ _LINE_ _, _ _STDC_ _, preprocessing

_exit() — Extended function (libc)

Terminate a program
int _exit(int *status***);**

_exit terminates a program directly. It returns *status* to the calling program, and exits.

Unlike the library function **exit**, **_exit** does not perform extra termination cleanup, such as flushing buffered files and closing open files.

_exit should be used only in situations where you do *not* want buffers flushed or files closed, such as when your program detects an irreparable error condition and you want to "bail out" to keep your data files from being corrupted.

_exit should also be used with programs that do not use STDIO and have been compiled with the **-ns** option to the **cc** command. Unlike **exit**, **_exit** does not use STDIO. This will help you create programs that are extremely small when compiled.

See Also

exit, extended miscellaneous, runtime startup, system

_tolower() — Extended macro (xctype.h)

Convert letter to lower case
#include <xctype.h>
int _tolower(int *c***);**

The macro **_tolower** converts *c* to lower case and returns it. If *c* is not a letter, the result is undefined.

_tolower differs from its cousin **tolower** in that **_tolower** is a macro that does not check whether its argument is in fact an alphanumeric character, whereas **tolower** is a function that does check its argument.

Example

This example opens a file of text and reverses the cases of all characters. It demonstrates **_tolower** and **_toupper**.

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <xctype.h>

void
fatal(char *message)
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      FILE *fp;
      int ch;

      if (--argc != 1)
            fatal("Usage: example filename");

      if ((fp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open file for reading");

      while ((ch = fgetc(fp)) != EOF) {
            if ((isascii(ch) != 0) && ch != '\r')
                  fatal("Not a text file");

            if (isalpha(ch) != 0)
                  fputc((isupper(ch) ? _tolower(ch) : _toupper(ch)),
                        stdout);
            else
                  fputc(ch, stdout);
      }
      return EXIT_SUCCESS;
}
```

See Also

_toupper, character handling, tolower

Notes

To conform to the ANSI Standard, this macro has been moved from the header **ctype.h** to the header **xctype.h**. This may require that some code be altered.

This macro is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

_toupper() — Extended macro (xctype.h)

Convert letter to upper case
#include <xctype.h>
_toupper(int *c***);**

The macro **_toupper** returns *c* converted to upper case. If *c* is not a letter, the result is undefined.

_toupper differs from its cousin **toupper** in that **_toupper** is a macro that does not check whether its argument is in fact an alphanumeric character, whereas **toupper** is a function that does check its argument.

Example

For an example of this routine, see the entry for **_tolower**.

See Also

_tolower, character handling, toupper

Notes

To conform to the ANSI Standard, this macro has been moved from the header **ctype.h** to the header **xctype.h**. This may require that some code be altered.

This macro is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

_zero() — i8086 support (libc)

Zero a block of memory
void _zero(unsigned *offs***, unsigned** *seg***, unsigned** *n***);**

_zero zeros out *n* bytes of memory at the address given by the segment *seg* and the offset *offs*.

_zero requires the full offset/segment address to work properly. If your program is compiled into SMALL model, you should use the macro **PTR** to ensure that a full address is used.

Example

The following example initializes a chunk of memory, displays it, and then zeroes it out.

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      char foo[80] = "Here is a string.";
      printf("Before _zero: %s\n", foo);

      _zero(PTR(foo), 80);
      printf("\nAfter _zero: %s\n", foo);
      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, PTR

{} — Punctuator

The punctuators {}, or “braces”, are used to delimit a block, and to group initializers. Braces must be used in pairs.

Cross-reference

Standard, §3.1.6

See Also

block, initialization, punctuators

| — Operator

Bitwise inclusive OR operator
operand1 **|** *operand2*

The operator **|** performs an bitwise inclusive OR operation. Each operand must have integral type. Each undergoes the usual arithmetic conversions, and the result has integral type.

A bitwise inclusive OR operation compares the bit patterns of the operands. It then sets each bit in the result if either, or both, of the corresponding bits in each of the operands is set.

For example, consider an environment which uses extended ASCII. Here, the character **9** is represented by the bit pattern

```
0011 1001
```

and the character **w** by the bit pattern:

```
0111 0111
```

Thus, the operation:

```
'9' | 'w'
```

yields the following bit pattern:

```
0000 0000 0111 1111
```

The extra "nybbles" to the left are created by the promotion of the character constants to type **int**.

The bitwise inclusive OR operation is also called the "union" of two bitsets.

Cross-references

Standard, §3.3.12
The C Programming Language, ed. 2, pp. 48, 207

See Also

^, |=, expressions

|= — Operator

Bitwise inclusive-OR assignment operator
operand1 **|=** *operand2*

The operator **|=** performs a bitwise inclusive OR operation on *operand1* and *operand2*, and assigns the result to *operand1*. It is equivalent to the expression

```
operand1 = operand1 | operand2
```

Both operands must have integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

|, expressions

|| — Operator

Logical OR operator
operand1 **||** *operand2*

The operator **||** performs a logical OR operation. Both *operand1* and *operand2* must have scalar type.

The result of the **||** operation has type **int**. The value of the result is one if either operand is true (nonzero); if both operands are false (equal to zero), the result has a value of zero.

The operands are evaluated from left to right. If *operand1* is true, then *operand2* is not evaluated. If *operand2* is an expression that yields a side-effect, the results of the **||** operation may not be what you expect: if *operand1* is true, *operand2* is not evaluated and its side-effect not generated.

LEXICON

Cross-references

Standard, §3.3.14
The C Programming Language, ed. 2, p. 208

See Also

&&, expressions

~ — Operator

Bitwise complement operator
~*operand*

The operator **~** is the bitwise complement operator. Its operand has an integral type, which undergoes integral promotion. The result is an object whose type is that of the promoted operand and whose bit pattern inverts that of the operand. This is also called a "one's complement operation".

For example, consider the object:

```
char example = 'a';
```

In an environment that uses extended ASCII, **example** will have the following bit pattern:

```
0110 0001
```

Thus, the expression **~example** promotes **example** to an **int**, and then generates an object with the following bit pattern:

```
1111 1111 1001 1110
```

As can be seen, the lower eight bits have been flipped. The eight bits on the left were added when the object was promoted to **int**. These new bits were initially set to zeroes when the character was promoted to an **int**, then the complement operation flipped the zeroes to ones. In this case, the sign bit is said to *propagate*.

Cross-references

Standard, §3.3.3.3
The C Programming Language, ed. 2, p. 204

See Also

!, expressions, integral promotion

A

abort() — General utility (libc)

End program immediately
void abort(void)

abort terminates a program's execution immediately. It is used to "bail out" of a program when a severe, unrecoverable problem occurs. It does not return.

abort terminates the program by calling **exit** with status **EXIT_FAILURE**.

abort prints the relative address from the beginning of the program, so that you can look the location up in the symbol table. See the entry for **nm** for more information on how to extract the symbol table from an executable program.

Example

This example simply aborts itself. For an example that uses **abort** in a more realistic manner, see **signal**.

```
#include <stdlib.h>
#include <stdio.h>

main(void)
{
      puts("...Dave ... I can feel my memory going ...");
      abort();
}
```

Cross-references

Standard, §4.10.4.1
The C Programming Language, ed. 2, p. 252

See Also

atexit, exit, general utilities, getenv, program termination, system

Notes

Some implementations of **abort**, specifically the one included with UNIX system V, permit it to return. The Standard forbids **abort** to return.

abs() — General utility (libc)

Compute the absolute value of an integer
#include <stdlib.h>
int abs(int *n***);**

abs returns the absolute value of integer *n*. The *absolute value* of a number is its distance from zero. This is *n* if *n***>=0**, and -*n* otherwise.

Example

This example checks whether **abs** is defined for all values on your implementation.

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
```

```
main(void)
{
      if(INT_MAX != abs(INT_MIN))
            printf("abs of %d is undefined\n", INT_MIN);
      return(EXIT_SUCCESS);
}
```

Cross-reference

Standard, §4.10.6.1
The C Programming Language, ed. 2, p. 253

See Also

div, general utilities, labs, ldiv

Notes

On two's complement machines, the absolute value of the most negative number may not be representable.

abs was originally declared in the header **math.h**. The Standard moved this function to **stdlib.h** on the grounds that it does not return **double**. This change may require that some existing code be altered.

access() — Access checking (libc)

Check if a file can be accessed in a given mode
#include <access.h>
int access(char ***filename***, int** *mode***);**

access checks whether a file can be accessed in the mode you wish. *filename* is the full path name of the file you wish to check. *mode* is the mode in which you wish to access *filename*, as follows:

1	AEXEC	Execute the file
2	AWRITE	Write into the file
4	AREAD	Read the file

The header **access.h** defines the manifest constants that are commonly used with **access**.

access returns zero if *filename* can be accessed in the requested mode, and a number greater than zero if it cannot.

Example

The following example checks if a file can be accessed in a particular manner.

```
#include <access.h>
#include <path.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(char *message)
{
      sprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}
```

```
main(int argc, char *argv[])
{
      char *env, *pathname;
      extern char *getenv(), *path();
      int mode;
      extern int access();

      if (argc != 3)
            fatal("Usage: access filename mode");

      switch(*argv[2]) {
            case 'e':
            case 'E':
                  mode = AEXEC;
                  break;
            case 'w':
            case 'W':
                  mode = AWRITE;
                  break;
            case 'r':
            case 'R':
                  mode = AREAD;
                  break;
            default:
                  fatal("modes: e=execute, w=write, r=read");
      }

      env = getenv("PATH");
      if ((pathname = path(env,argv[1],mode)) != NULL) {
            printf("PATH = %s\n", env);
            printf("pathname = %s\n", pathname);

            if (access(pathname, mode) == 0)
                  printf("%s accessible in mode %s\n",
                        pathname, argv[2]);
            else
                  printf("%s not accessible in mode %d\n",
                         pathname, mode);
      } else {
            printf("file %s of mode %d not found in path\n",
                   argv[1], mode);
            exit(EXIT_FAILURE);
      }
      return EXIT_SUCCESS;
}
```

See Also

access checking, access.h, path

Notes

access is included mainly for compatibility with the UNIX operating system. The only meaningful test that **access** can perform on the Atari ST is to check if a file is writable.

access.h — Header

Define manifest constants used by access()
#include <access.h>

access.h is a header file that defines the manifest constants used with the function **access**.

LEXICON

See Also

access, access checking, header

access checking — Overview

Let's C includes the following routines to check the access to a given file:

access.h
access Check if a file can be accessed in a given mode

path.h
path Build a path name for a file

stat.h
stat Find file attributes

These routines are not described in the ANSI Standard. Any program that uses any of them does not conform strictly to the Standard, and may not be portable to other compilers or environments.

See Also

Library

acos() — Mathematics (libm)

Calculate inverse cosine
#include <math.h>
double acos(double *arg***);**

acos calculates the inverse cosine of *arg*, which should be in the range of from -1.0 to 1.0. Any other argument will trigger a domain error.

acos returns the result, which is in the range of from zero to π radians.

Cross-references

Standard, §4.5.2.1
The C Programming Language, ed. 2, p. 251

See Also

asin, atan, atan2, cos, mathematics, sin, tan

address — Definition

An *address* designates a location in memory.

Example

The following prints the address and contents of a given byte of memory.

```
#include <stdio.h>
#include <stdlib.h>
main(void)
{
      char byte = 'a';
      /* Note use of the '%p' format specifier */
      printf("Address==%p  Contents==\"%c\"\n",
            &byte, byte);
      return EXIT_SUCCESS;
}
```

Cross-reference

The C Programming Language, ed. 2, p. 94

See Also

&, Definitions, pointer

alias — Definitions

An *alias* for an object is alternative way to access that object.

Because C uses pointers, it can be impossible for the translator to keep track of all possible aliases for an object. Often, the translator must use "worst-case aliasing assumptions" when memory is read. These assumption are explained below.

The Standard refers to aliasing in the section on expressions (3.3). It allows the translator to assume that the only way to reference a given object is by an object of the same type, a pointer to an object of that type, or by a character pointer. Type qualifiers and sign do not count in this situation. The reason a character pointer is assumed to point to any type of object is one of historical practice.

By making use of this information concerning types, a translator is said to make more favorable aliasing assumptions, and produce better code. For example, consider the following code fragment:

```
fn(int *ip, float *fp)
{
      int i;
      float f;

      ip = &i;    /* line 1 */
      *fp = f;    /* line 2 */
}
```

Normally in an assignment to a dereferenced pointer (line 2), the translator must assume that such a statement can overwrite the values of all global variables and the values of all local variables that have had their addresses taken.

Because **fp** is a pointer to **float**, the assignment to ***fp** need not invalidate the value of **i**. The translator must assume only that the current values of other **float**s may have been changed.

Any attempt to trick the translator, such as with a statement of the form

```
*fp = (float) i;
```

generates undefined behavior.

See Also

Definitions, type qualifier

alien — C keyword

Name a non-standard function

The **alien** declaration tells **Let's C** that the following function name is not a standard C function.

With the Mark Williams family of C compilers, **alien** indicates that a function uses the PL/M calling conventions. These differ from C in a number of ways. First, the calling sequence for PL/M pushes the leftmost argument first, whereas the calling sequence for C functions pushes the rightmost argument first. In addition, PL/M arguments are popped by the called function, whereas C arguments are popped by the calling function. Finally, when **Let's C** compiles a C function, it appends an underbar '_' to the end of the function's name.

Use of the **alien** keyword allows direct calls of most PL/M procedures and functions; that is, it can generate PL/M calls as well as C calls. For example,

```
extern alien plmfn();
```

LEXICON

declares **plmfn** to be a function that uses PL/M calling conventions. Of course, the types of the arguments to **plmfn** must correspond to the types of the arguments the PL/M functions expects.

To use the **alien** keyword in a program compiled with **Let's C**, you must compile the program using the **-VALIEN** option to the **cc** command.

See Also

C keywords, Language, statements

alignment — Definition

The term *alignment* refers to the fact that some environments require the addresses of certain data types to be evenly divisible by a certain integer. Different processors have different alignment requirements. For example, the Motorola 68000 requires that every **int** have an address that is even (i.e., that is evenly divisible by two). The translator must ensure that data objects are aligned properly so that fetches to memory will be performed efficiently and on the correct data types.

The environment may require that empty bytes of "padding" be inserted into structures to ensure that every type is aligned properly. For example, on the M68000 the following structure

```
struct example {
      char member1;
      int member2;
};
```

will actually consist of four bytes: one byte to hold the **char**, two bytes to hold the **int**, and between them, one byte of padding to ensure that the **int** is aligned properly. Often, the alignment of a **struct** member will be the maximum alignment required to align any of its members' data types.

Because different environments require different forms of alignment, a program that is intended to be portable should not assume that the members of a structure abut each other.

An object of type **char *** has the least strict alignment.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, p. 185

See Also

char, Definitions, struct

arena — Definition

An **arena** is the area of memory that is available for a program to allocate dynamically at run time. It consists of an area of memory that is divided into *allocated* and *unallocated* blocks. Normally, SMALL model programs cannot increase the size of the arena at run time; however, LARGE model programs can do so to a limited extent. The unallocated blocks together form the "free memory pool."

Portions of the arena can be allocated using the functions **malloc**, **calloc**, or **realloc**; returned to the free memory pool with **free**; or checked to see if they are allocated or not with **notmem**.

See Also

Definitions, extended STDIO, LARGE model, SMALL model, STDIO

argc — Definition

argc is the conventional name for the first argument to the function **main**. It is of type **int**. It gives the number of strings in the array pointed to by **argv**, which is the second argument to **main**.

By definition, the value of **argc** is never negative.

Cross-references

Standard, §2.1.2.2
The C Programming Language, ed. 2, p. 114

See Also

argv, Environment, envp, main

argument — Definition

An *argument* is an expression that appears between the parentheses of a function call or invocation of a function-like macro. Multiple arguments are separated by commas. For example, the following function call

```
example(arg1, arg2, arg3);
```

has three arguments.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, p. 201

See Also

conversions, Definitions, parameter

Notes

The Standard uses the term "argument" when it refers to the actual arguments of a function call or macro invocation. It uses the term "parameter" to refer to the formal parameters given in the definition of the function or macro.

argv — Definition

char *argv[];
argv is the conventional name for the second argument to the function **main**. It points to an array of pointers to type **char**. The strings to which **argv** points are passed by the host environment. Each may change the behavior of the program, and each may be modified by the program. Thus, the strings are called *program parameters*.

The number of pointers in the **argv** array is given by **argc**, which is the first argument to **main**. By definition, **argv[0]** always points to the name of the program. If the name is not available from the environment, then ***argv[0]** must be a null character. **argv[1]** through **argv[argc-1]** point to the set of program parameters; **argv[argc]** must be a null pointer.

Cross-references

Standard, §2.1.2.2
The C Programming Language, ed. 2, p. 114

See Also

argc, Environment, envp, main

array declarators — Definition

An *array declarator* declares an array. It can also establish the size of the array and cause storage to be allocated for it.

For example, consider the declaration:

```
int example[10];
```

The brackets '[ ]' establish that **example** is an array; the constant **10** establishes that **example** has ten elements. Thus, **example** is established to be an array of ten **int**s; memory is reserved for the ten members.

The constant expression that sets the size of an array must be an integral constant greater than zero. It must be known by translation phase 7 so the appropriate amount of storage can be allocated.

An array declarator may be empty; for example:

```
int example[];
```

In this case, **example** is an incomplete type. It will be completed when it is initialized.

Cross-references

Standard, §3.5.4.2
The C Programming Language, ed. 2, p. 216

See Also

[], declarators, initialization

Notes

For two array types to be compatible, the type of element in each, the number of dimensions in each, and the size of each corresponding dimension (except the first) must be identical.

as — Command

i8086 assembler
as [-bglx] [-o*file***]** *filename***.s ...**

as is a multipass assembler that will assemble functions written in i8086 assembly language. **as** will assemble programs into either SMALL or LARGE model, and will generate an object module in MS-DOS object format. It also supports i8087 opcodes, and it allows you to write functions in a model-independent manner.

as is not intended to be used for full-scale assembly-language programming; therefore, it does not include some of the more elaborate features found in full-fledged assemblers. For example, it has no facility for conditional compilation or user-defined macros. However, **Let's C** allows you to use preprocessor instructions to perform conditional assembly and expand macros. In addition, **as** optimizes branches to take advantage of short addressing forms, where the span of the branch permits.

File Names

All files of assembly language must have the suffix **.s** or **.m**. A **.s** file contains only assembly language, and may be assembled either directly by **as** using the command line shown below, or through the **cc** command. If you ask **as** to assemble a file that does not have the suffix **.s**, it will refuse to do so.

A file with the suffix **.m** is one that is passed through the C preprocessor **cpp** before it is assembled. These files *cannot* be assembled directly by **as**, but must be passed to the compiler controller **cc**,

which will first invoke **cpp** and then **as**. For example, to assemble the file **foo.m**, use the instruction

```
cc foo.m
```

This allows you to use preprocessor instructions that conditionalize code within a file; for example, the same file can contain code for SMALL model and LARGE model, with **cpp** selecting the correct code when you assemble the file. An example of a **.m** file is given below. For more information on **.m** files, see the Lexicon entry for **larges.h**.

Usage

To invoke **as** directly through MS-DOS, use the following command:

as [-bglsx] [-o *file***]** *filename***.s ...**

The named *files* are concatenated and the resulting object code is written either into the file specified by the **-o** option, or into the file **l.out** if the **-o** option is not used.

The other options are as follows:

-b Create a LARGE-model object module. This module hs two segments: *modname*_**code** and *modname*_**data**. By default, **as** creates an object module that is in SMALL model. See the Lexicon entry for **model** for more information on how these differ.

-g Give all symbols that are undefined at the end of the first pass the type **undefined external**, as though they had been declared with a **.globl** directive.

-l Generate a listing of your program. The listing is written to the standard output device; you can redirect it to a file or to the printer by using the '>' operator after the **as** command line.

-s Strip all non-global symbols from the symbol table. This option should be used with programs whose symbol tables are large enough to cause the linker **ld** to fail.

-x Strip all non-global symbols that begin with the character 'L' from the symbol table of the object module. This is a limited version of the **-s** option described above. It speeds up the linking of files by removing compiler-generated labels from the symbol table.

Lexical Conventions

Assembler tokens consist of identifiers (also called "symbols" or "names"), constants, and operators.

An *identifier* is a sequence of alphanumeric characters (including the period '.' and the underscore '_'). The first character must *not* be numeric. Only the first 16 characters of the name are significant; the remainder are quietly thrown away. Upper case and lower case are considered different. The machine instructions, assembly directives, and frequently used built-in symbols are in lower case.

The following lists the identifiers that represent the i8086 machine registers, which are predefined:

```
ax    sp    al    ah    cs
bx    bp    bl    bh    ds
cx    si    cl    ch    es
dx    di    dl    dh    ss
```

With regard to *constants*, the assembler uses the same syntax as the C compiler: A sequence of digits with a leading '0' is taken to be an octal constant. A sequence of digits with a leading '0x' is taken to be a hexadecimal constant; in this base, the letters 'A' through 'F' have the decimal values 10 through 15. Any strings of digits that do not begin with '0' are taken to be decimal constants.

A *character constant* consists of an apostrophe followed by an ASCII character. The constant's value is the ASCII code for the character, which is right-justified in the machine word. For example, an instruction to move the letter 'A' to the register **al** could be expressed in any of four equivalent

LEXICON

ways:

```
movb    al $0x41        / hexadecimal
movb    al $0101        / octal
movb    al $'A          / character
movb    al $65          / decimal
```

The dollar sign indicates an immediate operand.

A blank space can be represented either as 0x20 (its ASCII value in hexadecimal), or as an apostrophe followed by a space ('), which on the page or screen resembles an apostrophe alone.

as represents character constants with the following escape sequences:

\b	backspace	(0010)
\f	form feed	(0014)
\n	newline	(0012)
\r	carriage return	(0015)
\t	tab	(0011)
\v	vertical tab	(0013)
*****nnn*	octal value	(0*nnn*)

The semicolon character ';' indicates a line break. This character must be used at the end of a line in a **.m** file, because the ANSI definition of the C preprocessor assumes that multi-line macro definitions are always a single logical line.

In the ANSI preprocessor, a macro expansion always occupies no more than one line, no matter how many lines the definition or the actual parameters to the macro span; therefore, you must embed semicolons in macros that you want to expand to more than one line. For example,

```
#define enter(n) .globl n;n: push si; push di
```

will be treated by **as** as if it read

```
.globl n
n:    push si
      push di
```

The following gives a more readable form of the macro **enter**:

```
#define enter(n)  .globl n;\
n:  push si;\
    push di
```

Blanks and Tabs

Blanks and tab characters may be used freely between tokens, but not within identifiers. A blank or a tabulation character is required to separate adjacent tokens not otherwise separated, e.g., between an instruction opcode and its first operand.

Comments

Comments are introduced by a slash ('/') and continue until the end of the line. All characters in comments are ignored by the assembler.

Program Sections

as permits you to divide programs into sections, each corresponding to a functional area of the address space. **as** gives each program section its own location counter during assembly.

Under SMALL model, a program can have up to eight program sections, which are organized into three groups, as shown below:

code:	**shri**	shared instruction
	prvi	private instruction
	bssi	uninitialized instruction
data:	**prvd**	private data
	bssd	uninitialized data
	shrd	shared data
	strn	strings
tables:	**symt**	symbol table

All Mark Williams assemblers use the same set of sections. This contributes to the portability of programs between operating systems. Not all the sections are distinct under MS-DOS, however; the meanings of the sections under MS-DOS are as follows:

shri (shared instruction) is the same as **prvi** (private instruction); *shared* refers to the sharing of physical memory between two or more concurrent processes, and this capability does not exist under MS-DOS. **prvi** is used for all code generated by the C compiler.

There is no distinction between **shrd** and **prvd**. The latter is used by the compiler for all external and static data that are explicitly initialized in a C program.

bssi and **bssd** are initialized to zero. **Let's C** uses the **bssd** section for external or static data that are not initialized: the C language guarantees that these data are in fact initialized to zeros. **Let's C** does not use the **bssi** section.

The **strn** (strings) section is actually a special part of the data section, that **Let's C** uses to store string constants. It is synonymous with **prvd** under MS-DOS.

The **symt** section contains the symbol table used by the linker. Both the C compiler and the assembler generate symbol tables that go in this section.

In most cases, you need not worry about what all these program sections are, and can simply write code under the keywords **.prvi** or **.shri**, and write data under the keywords **.prvd** or **.shrd**. Do not to place items in the **symt** section, because the C compiler, the assembler, and the linker use it to communicate among themselves.

Under LARGE model, the assembled module has two sections: *filename*_**code** and *filename*_**data**. The former contains all code, that is,w hat goes into the **shri**, **prvi**, and **bssi** sections in SMALL model. The latter contains all data, that is, what goes into the **shrd**, **prvd**, **bssd**, and **strn** sections under SMALL model.

When a program is assembled, the sections of a program are concatenated so that in the assembly listing the whole program looks like a solid block of code and data. All code sections are combined into the i8086 **code** segment, and all data sections into the i8086 **data** segment. The symbol table is not actually linked when the program is executed, and so is not assigned to any i8086 segment

The Current Location

The special symbol '.' (dot) is a counter that represents the current location. The current location can be changed by an assignment; for example:

```
. = .+START
```

The assignment must not cause the value to decrease, and it must not change the program section, i.e., the right-hand operand must be defined in the same section as the current section.

Expressions

An expression is a sequence of symbols representing a value and a program section. Expressions are made up of identifiers, constants, operators, and brackets. All binary operators have equal precedence and are executed in a strict left-to-right order (unless altered by brackets).

LEXICON

Notice that brackets '[' and ']' group expression elements, because parentheses are used for indexed register addressing.

Types

Every expression has a type determined by its operands. The simplest operands are *symbols*. The following names the types of symbols available:

Undefined — A symbol is *defined* if it is a constant or a label, or if it is assigned a defined value; otherwise, it is *undefined*. A symbol may become undefined if it is assigned the value of an undefined expression. It is an error to assemble an undefined expression in pass 2. Pass 1 allows assembly of undefined expressions, but phase errors may be produced if undefined expressions are used in certain contexts, such as in a **.blkw** or **.blkb**.

Absolute — An *absolute* symbol is one defined ultimately from a constant or from the difference of two relocatable values.

Register — These are the machine registers.

Relocatable — All other user symbols are relocatable symbols in some program section. Each program section is a different relocatable type.

Any keyword may be used in an expression to obtain the basic value of the keyword. This may be useful when employing the keywords that define machine instructions. The basic value of a machine operation by default has the highest opcode associated with it; for example

```
.word push
```

yields **FF**.

Note that the type of an expression does not include such attributes as length (word or byte), so the assembler will not remember whether you defined a particular variable to be a word or a byte. Addresses and constants have different types, but the assembler does not treat a constant as an immediate value unless it is preceded by a dollar sign '$'. If you use a constant where an address is expected, **as** will treat the constant like an address (and vice versa). You must distinguish between variables and addresses or immediate values.

Operators

The following lists the operators that **as** recognizes:

+	addition
-	subtraction
*	multiplication
-	unary negation
~	unary complement
^	type transfer
|	segment construction

Expressions may be grouped with brackets. Parentheses are reserved for use in address mode descriptions.

Type propagation

When operands are combined in expressions, the resulting type is a function of both the operator and the types of the operands. The '*', '~', and unary '-' operators can only manipulate absolute operands and always yield an absolute result.

The '+' operator signifies the addition of two absolute operands to yield an absolute result, and the addition of an absolute to a relocatable operand to yield a result with the same type as the

relocatable operand.

The binary '-' operator allows two operands of the same type, including relocatable, to be subtracted to yield an absolute result; it also allows an absolute to be subtracted from a relocatable, to yield a result with the same type as the relocatable operand.

The binary operator '^' yields a result with the value of its left operand and the type of its right operand. It can be used to create expressions, usually used in an assignment statement, with any desired type.

Statements

A program consists of a sequence of statements separated by newlines or by semicolons. There are four kinds of statements: null statements, assignment statements, keyword statements, and machine instructions.

A statement can be proceded by any number of labels. There are two kinds of labels: *name* and *temporary*.

A *name* label consists of an identifier followed by a colon (**:**). The program section and value of the label are set to that of the current location counter. It is an error for the value of a label to change during an assembly. This most often happens when an undefined symbol is used to control a location counter adjustment.

A *temporary* label consists of a digit (" toQ ") followed by a colon ':'. It defines temporary symbols of the form '*n*f' and '*n*b', where '*n*' is the digit of the label. References of the form '*n*f' refer to the first temporary label '*n*:' forward from the reference; those of the form '*n*b' refer to the first temporary label '*n*:' backward from the reference. Such labels conserve symbol table space in the assembler.

A *null statement* is an empty line, or a line containing only labels or a comment. It can occur anywhere. **as** ignores it, except in the case of a label, which **as** gives the current value of the location counter.

An *assignment statement* consists of an identifier followed by an equal sign '=' and an expression. The value and program section of the identifier are set to that of the expression. Any symbol defined by an assignment statement may be redefined, either by another assignment statement or by a label. An assignment statement is equivalent to the **equ** keyword statement found in many assemblers.

Assembler directives

Assembler directives allow you to pass instructions directly to **as**. Each directive begins with a period, and most are followed by operands.

The following describes the directives that **as** recognizes:

.ascii *string*

The first non-white space character, typically a quotation mark, that appears after the keyword is taken as a delimiter. Successive characters are assembled into successive bytes until until the delimiter appears again. To include a quotation mark within a string, use another character for the delimiter.

It is an error if a newline is encountered before reaching the second delimiter. To insert a newline into a string, use the character constant '\n', a described above.

.blkb/.blkw

Assemble blocks of bytes or words that are filled with zeroes. The size of the block is *expression* bytes or words.

LEXICON

.bssd Change the current program section to **bssd**. The current location is reset to the value of the **bssd** location counter.

.bssi Change the current program section to **bssi**. The current location is reset to the value of the **bssi** location counter.

.byte The *expression*s in the list are truncated to byte size and assembled into successive bytes. Expressions in the list are separated by commas.

.even/.odd
These insert a NULL byte, if necessary, to set the location counter to the next even or odd location, respectively. They are used to force alignment.

.globl The identifiers in the comma-separated list are marked as global. If they are defined in the current assembly, they may be referenced by other object modules; if they are undefined, they must be resolved by the linker before execution.

.page Force the printed listing of your assembly-language program to skip to the top of a new page by inserting a form-feed character into the file. The title is printed at the top of the page.

.prvd Change the current program section to **prvd**. The current location is reset to the value of the **prvd** location counter.

.prvi Change the current program section to **prvi**. The current location is reset to the value of the **prvi** location counter.

.shrd Change the current program section to **shrd**. The current location is reset to the value of the **shrd** location counter.

.shri Change the current program section to **shri**. The current location is reset to the value of the **shri** location counter.

.strn Change the current program section to **strn**. The current location is reset to the value of the **strn** location counter.

.title *string*
Print *string* at the top of every page in the listing. This directive also causes the listing to skip to a new page.

.word *expression* **[,** *expression* **]**
Truncate *expression*s to word length and assemble the resulting data into successive words. Expressions in the list are separated by commas.

Address descriptors

The source and destination descriptors use the following syntax. *r* refers to a register and the symbol *e* to an expression, as follows:

r: register
al, cl, dl, bl, ah, ch, dh, bh
ax, cx, dx, bx, sp, bp, si, di

e: direct address|
Any eight- or 16-bit number. Eight-bit numbers are sign extended.

(*r*): indexing
(si) (di) (bx)

f*e*(*r*): index displacement
e(si) *e*(di) *e*(bx): default segment is **ds**
e(bp): default segment is **ss**

(*r,r*): double index
(bx), si) (bx, di): default segment is **ds**
(bp, si) (bp, di): default segment is **ss**

e(*r,r*): double index with displacement
e(bx, si) *e*(bx, di): default segment is **ds**
e(bp, si) *e*(bp, di): default segment is **ss**

R*e*: immediate

s: segment register|
ss, ds, es, cs: allowed only where explicitliy stated.

Note that the dollar sign is always used to indicate an immediate value, even if the expression is a constant.

A direct address is interpreted as either a direct address or a PC-relative displacement, depending on the requirements of the instruction.

If an address descriptor indicates an indexing mode and the base expression is of type absolute, **as** uses the shortest displacement length (zero, one, or two bytes) that can hold the expression's value. Relocatable base expressions, whose values cannot be completely determined until the program is linked, are always assigned two-byte displacements.

Any address descriptor may be modified by a segment escape prefix. A segment escape prefix consists of a segment register name followed by a colon ':'. The escape causes **as** to produce a segment override prefix that uses the specified segment register as an operand. **as** does not produce segment override prefixes unless explicitly required by an instruction.

Instructions

The following machine instructions are defined. The examples illustrate the general syntax of the operands. Combinations that are syntactically valid may be forbidden for semantic reasons.

The examples use the following references:

a	general address
al	al register
ax	ax register
cl	cl register
d	direct address
dx	dx register
e	expression
$e	immediate expression
m	memory address (not an immediate)
p	port address

as treats as ordinary one-byte machine operations some operations that the Intel assembler ASM86 handles with special syntax; these include the *lock* and *repeat* prefixes. **as** makes no attempt to prevent the generation of incorrect sequences of these prefix bytes.

Although every machine operation has a type and value associated with it, in most cases the value was chosen to help **as** format the machine instructions.

For more information on these instructions, see the Intel *ASM86 Assembly Language Reference Manual*.

aaa	ASCII adjust AL after addition
aad	ASCII adjust AX before division
aam	ASCII adjust AX after multiply
aas	ASCII adjust AL after subtraction

LEXICON

adcb	*r, a*	Add with carry, byte
adc	*r, a*	Add with carry, word
adcb	*a, r*	Add with carry, byte
adc	*a, r*	Add with carry, word
adcb	*a, $e*	Add with carry, byte
adc	*a, $e*	Add with carry, word
addb	*r, a*	Add, byte
add	*r, a*	Add, word
addb	*a, r*	Add, byte
add	*a, r*	Add, word
addb	*a, $e*	Add, byte
add	*a, $e*	Add, word
andb	*r, a*	Logical and, byte
and	*r, a*	Logical and, word
andb	*a, r*	Logical and, byte
and	*a, r*	Logical and, word
andb	*a, $e*	Logical and, byte
and	*a, $e*	Logical and, word
call	*d*	Near call, PC-relative
cbw		Convert byte into word
clc		Clear carry flag
cld		Clear direction flag
cli		Clear interrupt flag
cmc		Complement carry flag
cmpb	*r, a*	Compare two operands, byte
cmp	*r, a*	Compare two operands, word
cmpb	*a, r*	Compare two operands, byte
cmp	*a, r*	Compare two operands, word
cmpb	*a, $e*	Compare two operands, byte
cmp	*a, $e*	Compare two operands, word
cmps		Compare string operands, bytes
cmpsb		Compare string operands, bytes
cmpsw		Compare string operands, words
cwd		Convert word to double
daa		Decimal adjust AL after addition
das		Decimal adjust AL after subtraction
decb	*a*	Decrement by one, byte
dec	*a*	Decrement by one, word
divb	*m*	Unsigned divide, byte
div	*m*	Unsigned divide, word
esc	*a*	Escape **0xD8**
hlt		Halt
icall	*a*	Near call, absolute offset at EA word
idivb	*m*	Signed divide, byte
idiv	*m*	Signed divide, word
ijmp	*a*	Jump short, absolute offset at EA word
imulb	*m*	Signed multiply, byte
imul	*m*	Signed multiply, word
inb	*al, p*	Input, byte
in	*ax, p*	Input, word
inb	*al, dx*	Input, byte
in	*ax, dx*	Input, word
incb	*a*	Increment by one, byte
inc	*a*	Increment by one, word

int	*e*	Call to interrupt
into		Call to interrupt, overflow
iret		Interrupt return
ja	*d*	Jump short if greater
jae	*d*	Jump short if greater or equal
jb	*d*	Jump short if less
jbe	*d*	Jump short if less or equal
jc	*d*	Jump short if carry
jcxz	*d*	Jump short if CX equals zero
je	*d*	Jump short if equal to
jg	*d*	Jump short if greater
jge	*d*	Jump short if greater or equal
jl	*d*	Jump short if less
jle	*d*	Jump short if less or equal
jmp	*d*	Jump short, PC-relative word offset
jmpb	*d*	Jump short, PC-relative byte offset
jmpl	*d*	Jump long
jna	*d*	Jump short if not above
jnae	*d*	Jump short if not above or equal
jnb	*d*	Jump short if not below
jnbe	*d*	Jump short if not below or equal
jnc	*d*	Jump short if not carry
jne	*d*	Jump short if not equal
jng	*d*	Jump short if not greater
jnge	*d*	Jump short if not greater or equal
jnl	*d*	Jump short if not less
jnle	*d*	Jump short if not less or equal
jno	*d*	Jump short if not overflow
jnp	*d*	Jump short if not parity
jns	*d*	Jump short if not sign
jnz	*d*	Jump short if not zero
jo	*d*	Jump short if overflow
jp	*d*	Jump short if parity
jpe	*d*	Jump short if parity even
jpo	*d*	Jump short if parity odd
js	*d*	Jump short if sign
jz	*d*	Jump short if zero
lahf		Load flags into AH register
lds	*r, a*	Load double pointer into DS
lea	*r, a*	Load effective address offset
les	*r, a*	Load double pointer into ES
lock		Assert BUS LOCK signal
lodsb		Load byte into AL
lods		Load byte into AL
lodsw		Load byte into AL
loop	*d*	Loop; decrement CX, jump short if CX less than zero
loope	*d*	Loop; decrement CX, jump short if CZ not zero and equal
loopne	*d*	Loop; decrement CX, jump short if CX not zero and not equal
loopnz	*d*	Loop; decrement CX, jump short if CZ not zero and ZF equals zero
loopz	*d*	Loop; decrement CX, jump short if CX not zero and zero
movb	*r, a*	Move, byte
mov	*r, a*	Move, word
movb	*a, r*	Move, byte
mov	*a, r*	Move, word

LEXICON

movb	*a, $e*	Move, byte
mov	*a, $e*	Move, word
movb	*a, s*	Move, byte
mov	*a, s*	Move, word
movb	*s, a*	Move, byte
mov	*s, a*	Move, word
movsb		Move string byte-by-byte
movs		Move string word-by-word
movsw		Move string word-by-word
mulb	*m*	Multiply, byte
mul	*m*	Multiply, word
negb	*a*	Two's complement negation, byte
neg	*a*	Two's complement negation, word
nop		No operation
notb	*a*	One's complement negation, byte
not	*a*	One's complement negation, word
orb	*r, a*	Logical inclusive OR, byte
or	*r, a*	Logical inclusive OR, word
orb	*a, r*	Logical inclusive OR, byte
or	*a, r*	Logical inclusive OR, word
orb	*a, $e*	Logical inclusive OR, byte
or	*a, $e*	Logical inclusive OR, word
outb	*p, al*	Output to port, byte
out	*p, ax*	Output to port, word
outb	*dx, al*	Output to port, byte
out	*dx, ax*	Output to port, word
pop	*m*	Pop a word from the stack
pop	*s*	Pop a word from the stack
popf		Pop fom stack into flags register
push	*m*	Push a word onto the stack
push	*s*	Push a word onto the stack
pushf		Push flags register onto the stack
rclb	*a, $1*	Rotate left $1 times, byte
rclb	*a, cl*	Rotate left CL times, byte
rcl	*a, $1*	Rotate left $1 times, word
rcl	*a, cl*	Rotate left CL times, word
rcrb	*a, $1*	Rotate right $1 times, byte
rcrb	*a, cl*	Rotate right CL times, byte
rcr	*a, $1*	Rotate right $1 times, word
rcr	*a, cl*	Rotate right CL times, word
rep		Repeat following string operation
repe		Find nonmatching bytes
repne		Repeat, not equal
repnz		Repeat, not equal
repz		Repeat, equal
ret		Return from procedure
rolb	*a, $1*	Rotate left, byte
rolb	*a, cl*	Rotate left, byte
rol	*a, $1*	Rotate left, word
rol	*a, cl*	Rotate left, word
rorb	*a, $1*	Rotate right, byte
rorb	*a, cl*	Rotate right, byte
ror	*a, $1*	Rotate right, word
ror	*a, cl*	Rotate right, word

sahf		Store AH into flags
salb	*a, $1*	Shift left, byte
salb	*a, cl*	Shift left, byte
sal	*a, $1*	Shift left, word
sal	*a, cl*	Shift left, word
sarb	*a, $1*	Shift right, byte
sarb	*a, cl*	Shift right, byte
sar	*a, $1*	Shift right, word
sar	*a, cl*	Shift right, word
sbbb	*r, a*	Integer subtract with borrow, byte
sbb	*r, a*	Integer subtract with borrow, word
sbbb	*a, r*	Integer subtract with borrow, byte
sbb	*a, r*	Integer subtract with borrow, word
sbbb	*a, $e*	Integer subtract with borrow, byte
sbb	*a, $e*	Integer subtract with borrow, word
scasb		Compare string data, byte
scas		Compare string data, word
shlb	*a, $1*	Shift left, byte
shlb	*a, cl*	Shift left, byte
shl	*a, $1*	Shift left, word
shl	*a, cl*	Shift left, word
shrb	*a, $1*	Shift right, byte
shrb	*a, cl*	Shift right, byte
shr	*a, $1*	Shift right, word
shr	*a, cl*	Shift right, word
stc		Set carry flag
std		Set direction flag
sti		Set interrupt enable flag
stosb		Store string data, byte
stos		Store string data, byte or word
stosw		Store string data, word
subb	*r, a*	Integer subtraction, byte
sub	*r, a*	Integer subtraction, word
subb	*a, r*	Integer subtraction, byte
sub	*a, r*	Integer subtraction, word
subb	*a, $e*	Integer subtraction, byte
sub	*a, $e*	Integer subtraction, word
testb	*r, a*	Logical compare, byte
test	*r, a*	Logical compare, word
testb	*a, r*	Logical compare, byte
test	*a, r*	Logical compare, word
testb	*a, $e*	Logical compare, byte
test	*a, $e*	Logical compare, word
wait		Wait until BUSY pin is inactive
xcall	*d, d*	Far call, immediate four-byte address
xchgb	*r, a*	Exchange memory, byte
xchg	*r, a*	Exchange memory, word
xicall		Far call, address at EA double word
xijmp		Jump far, address at memory double word
xjmp	*d, d*	Jump far, immediate four-byte address
xlat		Table look-up translation
xorb	*r, a*	Logical exclusive OR, byte
xor	*r, a*	Logical exclusive OR, word
xorb	*a, r*	Logical exclusive OR, byte

xor	*a, r*	Logical exclusive OR, word
xorb	*a, $e*	Logical exclusive OR, byte
xor	*a, $e*	Logical exclusive OR, word
xret		Return, intersegment

i8087 instructions

as can also generate object files that use the i8087 mathematics co-processor. The example instructions use the following references:

d	direct address
st0	floating point register 0
st1	any floating point register *except* 0

The following lists the i8087 instructions:

fabs		Absolute value
fadd	*st0, st1*	Add real
fadd	*st1, st0*	Add real
ffadd	*d*	Add real, float
fdadd	*d*	Add real, double
faddp		Add real and pop
faddp	*st, st0*	Add real and pop
fbld	*d*	Load packed decimal (BCD)
fbstp	*d*	Store packed decimal (BCD) and pop
fchs		Change sign
fclex		Clear exception
fnclex		Clear exception
fcom		Compare real
ffcom	*d*	Compare real, float
fdcom	*d*	Compare real, double
fcomp		Compare real and pop
fcomp	st1	Compare real and pop
ffcomp	*d*	Compare real and pop, float
fdcomp	*d*	Compare real and pop, double
fcompp		Compare real and pop twice
fdecstp		Decrement stack pointer
fdisi		Disable interrupts
fndisi		Disable interrupts, no operands
fdiv	*st0, st1*	Divide real
fdiv	*st1, st0*	Divide real
ffdiv	*d*	Divide real, float
fddiv	*d*	Divide real, double
fdivp		Divide real and pop
fdivp	st1	Divide real and pop
fdivr	*st0, st1*	Divide real reversed
fdivr	*st1, st0*	Divide real reversed
ffdivr	*d*	Divide real reversed, float
fddivr	*d*	Divide real reversed, double
fdivrp		Divide real reversed and pop
fdivrp	*st1*	Divide real reversed and pop
feni		Enable interrupts
fneni		Enable interrupts, no operands
ffree	*st1*	Free register
fiadd	*d*	Integer add
fladd	*d*	Integer add, long
ficom	*d*	Integer compare

ficom	*d*	Integer compare, long
ficomp	*d*	Integer compare and pop
ficomp	*d*	Integer compare and pop, long
fidiv	*d*	Integer divide
fldiv	*d*	Integer divide, long
fidivr	*d*	Integer divide reversed
fldivr	*d*	Integer divide, long reversed
fild	*d*	Integer load
flld	*d*	Integer load, long
fqld	*d*	Integer load, quad
fimul	*d*	Integer multiply
flmul	*d*	Integer multiply, long
fincstp		Increment stack pointer
finit		Initialize processor
fninit		Initialize processor
fist	*d*	Integer store
flst	*d*	Integer store, long
fistp	*d*	Integer store and pop
flstp	*d*	Integer store and pop, long
fqstp	*d*	Integer store and pop, quad
fisub	*d*	Integer subtract
flsub	*d*	Integer subtract, long
fisubr	*d*	Integer subtract reversed
flsubr	*d*	Integer subtract reversed, long
fld	*st1*	Load real
ffld	*d*	Load real, float
fdld	*d*	Load real, double
ftld	*d*	Load real, temp
fldcw	*d*	Load control word
fldenv	*d*	Load environment
fldlg2		Load log(10)2
fldln2		Load log(e)2
fldl2e		Load log(2)e
fldl2t		Load log(2)10
fldpi		Load pi
fldz		Load +0.0
fld1		Load +1.0
fmul		Multiply real
fmul	*st0, st1*	Multiply real
ffmul	*st1, st0*	Multiply real, float
fdmul	*d*	Multiply real, double
fmulp	*d*	Multiply real and pop
fnop	st1	No operation
fpatan		Partial arctangent
fprem		Partial remainder
fptan		Partial tangent
frndint		Round to integer
frstor	*d*	Restore saved state
fsave	*d*	Save state
fnsave	*d*	Save state
fscale		Scale
fsetpm		Set protected mode
fsqrt		Square root
fst	*st1*	Store real

LEXICON

ffst	*d*	Store real, float
fdst	*d*	Store real, double
fstcw	*d*	Store control word
fnstcw	*d*	Store control word
fstenv	*d*	Store environment
fnstenv	*d*	Store environment
fstp	*st1*	Store real and pop
ffstp	*d*	Store real and pop, float
fdstp	*d*	Store real and pop, double
ftstp	*d*	Store real and pop, temp
fstsw	*d*	Store status word
fnstsw	*d*	Store status word
fsub	*st0, st1*	Subtract real
fsub	*st1, st0*	Subtract real
ffsub	*d*	Subtract real, float
fdsub	*d*	Subtract real, double
fsubp		Subtract real and pop
fsubp	*st1*	Subtract real and pop
fsubr	*d*	Subtract real reversed
ffsubr	*d*	Subtract real reversed, float
fdsubr	*d*	Subtract real reversed, double
fsubrp		Subtract real reversed and pop
fsubrp	*st1*	Subtract real reversed and pop
ftst		Test stack top against +0.0
fwait		Wait while 8087 is busy
fxam		Examine stack top
fxch	*st1*	Exchange registers
fxch		Exchange registers
fxtract		Extract exponent and significance
fyl2x		Y*log(2)X
fyl2xp1		Y*log(2)(X+1)

Examples

The first example executes the program **hello.c** in a model-independent assembly language. If executed, it should be placed in a file called **hello.m**, and assembled through the **cc** command, as follows:

```
cc -o hello hello.m
```

The **cc** command will pass the program first to the C preprocessor **cpp**, and then to **as**. For more information, see the Lexicon entry for **larges.h**.

```
#include <larges.h>
     .prvd
Hi:  .ascii     "Hello world.\n"
     .shri
     Enter(main_)      /* Note use of C-style comments */
     mov  ax, $Hi      /* push offset of msg */
     push ax
#ifdef LARGEDATA
     mov  ax, $@Hi     /* push segment of msg */
     push ax
#endif
     Gcall    printf_
     add sp, $RASIZE
     Leave
```

The next example program, **strchar.s** defines a function **strchar** that returns the number of occurrences of a character in a string.

FILE: strchar.s

```
/
/
/  Count and return the occurrences
/  of a character in a string.
/
/     int
/     strchar(s, c)
/     char *s;
/     int c;
/
/

  .globl     strchar_   / Make the name known externally.

  strchar_:
       push    si         / Standard C function
       push    di         / linkage. Save the
       push    bp         / si, di, and bp registers
       mov     bp, sp     / and set up new frame pointer.

       mov     si, 8(bp)  / String ptr -> si.
       mov     bx, 10(bp) / Char -> bx (actually bl).
       sub     ax, ax     / Clear ax (count register).
       sub     cx, cx     / Clear cx.

  0:   movb    cl, (si)   / Get character from string.
       jcxz    2f         / End of string?
       cmpb    bl, cl     / No. Do chars match?
       jnz     1f         / No.
       inc     ax         / Yes. Increment count.

  1:   inc     si         / Bump string pointer
       jmp     0b         / and loop again.

  2:   pop     bp         / Standard C return
       pop     di         / linkage. Restore
       pop     si         / saved registers and
       ret                / go home.
```

The following C program, **main.c** uses **strchar** The assembly language listing that follows, **main.s** was produced from **main.c** by the **-VASM** option in **cc**. The listing has been edited, and comments added, to illustrate what is happening.

```
/* FILE: main.c */

      main()
      {
          int n;
          n = strchar("aardvark", 'a');
      }

            .shri               / ''code'' program section.

            .globl   main_

      main_:

            .strn               / ''string'' program section.
```

LEXICON

```
L2:   .byte    0x61        / This is the string
      .byte    0x61        / ''aardvark''
      .byte    0x72
      .byte    0x64
      .byte    0x76
      .byte    0x61
      .byte    0x72
      .byte    0x6B
      .byte    0x00

      .shri                / Back to ''code''

      push  si             / Standard C function
      push  di             / linkage.  Save registers,
      push  bp             / set up new frame pointer (bp),
      mov   bp, sp         / and make room on stack
      sub   sp, $0x02      / for the auto int, ''n''

      mov   ax, $0x61      / Push the
      push  ax             / character 'a'.
      mov   ax, $L2        / Push the address
      push  ax             / of the string ''aardvark''
      call  strchar_       / Function call.
      add   sp, $0x04      / Remove args from stack.
      mov   -0x02(bp), ax  / Assign result to auto 'n'.

      mov   sp, bp         / Standard C return
      pop   bp             / linkage.  Adjust stack
      pop   di             / pointer, then restore
      pop   si             / registers and
      ret                  / go home.
```

See Also

C language, calling conventions, cc, larges.h, memory allocation

ASCII — Definition

ASCII is an acronym for the American Standard Code for Information Interchange. It is a table of seven-bit binary numbers that encode the letters of the alphabet, numerals, punctuation, and the most commonly used control sequences for printers and terminals.

The extended ASCII character set defines eight-bit encodings. The lower 127 characters are those of standard ASCII, and the higher 127 characters are also defined.

Though the standard ASCII character set is used commonly throughout the United States, other countries use the ISO 646 character set, which is an invariant subset of standard ASCII. See the entry on **trigraphs** for a discussion of the representing C characters in environments in which not all of the 127 ASCII characters are available.

The following table gives the lower 127 ASCII characters in octal, decimal, and hexadecimal numbers.

000	0	0x00	NUL	**<ctrl-@>**	Null character
001	1	0x01	SOH	**<ctrl-A>**	Start of header
002	2	0x02	STX	**<ctrl-B>**	Start of text
003	3	0x03	ETX	**<ctrl-C>**	End of text
004	4	0x04	EOT	**<ctrl-D>**	End of transmission
005	5	0x05	ENQ	**<ctrl-E>**	Enquiry
006	6	0x06	ACK	**<ctrl-F>**	Positive acknowledgement
007	7	0x07	BEL	**<ctrl-G>**	Alert
010	8	0x08	BS	**<ctrl-H>**	Backspace

011	9	0x09	HT	**<ctrl-I>**	Horizontal tab
012	10	0x0A	LF	**<ctrl-J>**	Line feed
013	11	0x0B	VT	**<ctrl-K>**	Vertical tab
014	12	0x0C	FF	**<ctrl-L>**	Form feed
015	13	0x0D	CR	**<ctrl-M>**	Carriage return
016	14	0x0E	SO	**<ctrl-N>**	Shift out
017	15	0x0F	SI	**<ctrl-O>**	Shift in
020	16	0x10	DLE	**<ctrl-P>**	Data link escape
021	17	0x11	DC1	**<ctrl-Q>**	Device control 1 (XON)
022	18	0x12	DC2	**<ctrl-R>**	Device control 2 (tape on)
023	19	0x13	DC3	**<ctrl-S>**	Device control 3 (XOFF)
024	20	0x14	DC4	**<ctrl-T>**	Device control 4 (tape off)
025	21	0x15	NAK	**<ctrl-U>**	Negative acknowledgement
026	22	0x16	SYN	**<ctrl-V>**	Synchronize
027	23	0x17	ETB	**<ctrl-W>**	End of transmission block
030	24	0x18	CAN	**<ctrl-X>**	Cancel
031	25	0x19	EM	**<ctrl-Y>**	End of medium
032	26	0x1A	SUB	**<ctrl-Z>**	Substitute
033	27	0x1B	ESC	**<ctrl-[>**	Escape
034	28	0x1C	FS	**<ctrl-\>**	Form separator
035	29	0x1D	GS	**<ctrl-]>**	Group separator
036	30	0x1E	RS	**<ctrl-^>**	Record separator
037	31	0x1F	US	**<ctrl-_>**	Unit separator
040	32	0x20	SP	Space	
041	33	0x21	!	Exclamation point	
042	34	0x22	"	Quotation mark	
043	35	0x23	#	Pound sign (sharp)	
044	36	0x24	$	Dollar sign	
045	37	0x25	%	Percent sign	
046	38	0x26	&	Ampersand	
047	39	0x27	'	Apostrophe	
050	40	0x28	(	Left parenthesis	
051	41	0x29	)	Right parenthesis	
052	42	0x2A	*	Asterisk	
053	43	0x2B	+	Plus sign	
054	44	0x2C	,	Comma	
055	45	0x2D	-	Hyphen (minus sign)	
056	46	0x2E	.	Period	
057	47	0x2F	/	Virgule (slash)	
060	48	0x30	0		
061	49	0x31	1		
062	50	0x32	2		
063	51	0x33	3		
064	52	0x34	4		
065	53	0x35	5		
066	54	0x36	6		
067	55	0x37	7		
070	56	0x38	8		
071	57	0x39	9		
072	58	0x3A	:	Colon	
073	59	0x3B	;	Semicolon	
074	60	0x3C	<	Less-than symbol (left angle bracket)	
075	61	0x3D	=	Equal sign	
076	62	0x3E	>	Greater-than symbol (right angle bracket)	

077	63	0x3F	?	Question mark
0100	64	0x40	@	At sign
0101	65	0x41	A	
0102	66	0x42	B	
0103	67	0x43	C	
0104	68	0x44	D	
0105	69	0x45	E	
0106	70	0x46	F	
0107	71	0x47	G	
0110	72	0x48	H	
0111	73	0x49	I	
0112	74	0x4A	J	
0113	75	0x4B	K	
0114	76	0x4C	L	
0115	77	0x4D	M	
0116	78	0x4E	N	
0117	79	0x4F	O	
0120	80	0x50	P	
0121	81	0x51	Q	
0122	82	0x52	R	
0123	83	0x53	S	
0124	84	0x54	T	
0125	85	0x55	U	
0126	86	0x56	V	
0127	87	0x57	W	
0130	88	0x58	X	
0131	89	0x59	Y	
0132	90	0x5A	Z	
0133	91	0x5B	[	Left bracket (left square bracket)
0134	92	0x5C	\	Backslash
0135	93	0x5D	]	Right bracket (right square bracket)
0136	94	0x5E	^	Circumflex
0137	95	0x5F	_	Underscore (underbar)
0140	96	0x60	`	Grave
0141	97	0x61	a	
0142	98	0x62	b	
0143	99	0x63	c	
0144	100	0x64	d	
0145	101	0x65	e	
0146	102	0x66	f	
0147	103	0x67	g	
0150	104	0x68	h	
0151	105	0x69	i	
0152	106	0x6A	j	
0153	107	0x6B	k	
0154	108	0x6C	l	
0155	109	0x6D	m	
0156	110	0x6E	n	
0157	111	0x6F	o	
0160	112	0x70	p	
0161	113	0x71	q	
0162	114	0x72	r	
0163	115	0x73	s	
0164	116	0x74	t	

0165	117	0x75	u	
0166	118	0x76	v	
0167	119	0x77	w	
0170	120	0x78	x	
0171	121	0x79	y	
0172	122	0x7A	z	
0173	123	0x7B	{	Left brace (left curly bracket)
0174	124	0x7C	|	Vertical bar
0175	125	0x7D	}	Right brace (right curly bracket)
0176	126	0x7E	~	Tilde
0177	127	0x7F	DEL	Delete

See Also

Definitions, trigraph sequences

asctime() — Time function (libc)

Convert broken-down time to text
#include <time.h>
char *asctime(const struct tm ****timestruct*);**

The function **asctime** converts the data pointed to by *timestruct* into a text string of the form:

```
Wed Dec 10 13:57:33 1987\n\0
```

The structure pointed to by *timestruct* must first be initialized by either the function **gmtime** or the function **localtime** before it can be used by **asctime**. See the entry for **tm** for further information on this structure.

asctime returns a pointer to the string it creates.

Example

This example uses **asctime** to display Universal Coordinated Time.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      printf(asctime(gmtime(NULL)));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.3.1
The C Programming Language, ed. 2, p. 256

See Also

ctime, date and time, gmtime, localtime, strftime, time_t, tm

Notes

asctime writes its string into a static buffer that will be written by another call to either **asctime** or **ctime**.

The name “asctime” is short for “ASCII time”; its use, however, is not limited to implementations on ASCII systems.

The Standard describes the following algorithm with which **asctime** can generate its string:

LEXICON

```
char *
asctime(const struct tm *timeptr)
{
      static const char wday_name[7][3] = {
            "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
      };
      static const char mon_name[12][3] = {
            "Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
      };
      static char result[26];

      sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
            wday_name[timeptr->tm_wday],
            mon_name[timeptr->tm_mon],
            timeptr->tm_mday, timeptr->tm_hour,
            timeptr->tm_min, timeptr->tm_sec,
            1900 + timeptr->tm_year);

      return result;
}
```

asin() — Mathematics (libm)

Calculate inverse sine
#include <math.h>
double asin(double *arg***);**

asin calculates the inverse sine of *arg*, which must be in the range of from -1.0 to 1.0; any other value will trigger a domain error.

asin returns the result, which is in the range $\pi/2$ to π.

Cross-references

Standard, §4.5.2.2
The C Programming Language, ed. 2, p. 251

See Also

acos, atan, atan2, cos, mathematics, sin, tan

assert() — Diagnostics (assert.h)

Check assertion at run time
#include <assert.h>
void assert(int *expression***);**

assert checks the value of *expression*. If *expression* is false (zero), **assert** sends a message into the standard error stream and calls **abort**. It is useful for verifying that a necessary condition is true.

The error message includes the text of the assertion that failed, the name of the source file, and the line within the source file that holds the expression in question. These last two elements consist, respectively, of the values of the preprocessor macros _ _**FILE**_ _ and _ _**LINE**_ _.

Because **assert** calls **abort**, it never returns.

To turn off **assert**, define the macro **NDEBUG** prior to including the header **assert.h**. This forces **assert** to be redefined as

```
#define assert(ignore)
```

Example

This program generates an error if your implementation does not conform to the Standard.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

main(void)
{
#ifdef STDC
      assert(STDC);
#else
      fprintf(stderr, "Not ANSI C\n");
#endif
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.2.1.1
The C Programming Language, ed. 2, p. 253

See Also

abort, assert.h, diagnostics, NDEBUG

Notes

The Standard requires that **assert** be implemented as a macro, not a library function. If a program suppresses the macro definition in favor of a function call, its behavior is undefined.

Turning off **assert** with the macro **NDEBUG** will affect the behavior of a program if the expression being evaluated normally generates side effects.

assert is useful for debugging, and for testing boundary conditions for which more graceful error recovery has not yet been implemented.

assert.h — Header

Header for assertions
#include <assert.h>

assert.h is the header file that defines the macro **assert**.

Cross-references

Standard, §4.2
The C Programming Language, ed. 2, pp

See Also

assert, diagnostics, header

atan() — Mathematics (libm)

Calculate inverse tangent
#include <math.h>
double atan(double *arg***);**

atan calculates the inverse tangent of *arg*, which may be any real number.

atan returns the result, which is in the range of from $-\pi/2$ to $\pi/2$ radians.

Cross-references

Standard, §4.5.2.3
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan2, cos, mathematics, sin, tan

atan2() — Mathematics (libm)

Calculate inverse tangent
#include <math.h>
double atan2(double *num*, **double** *den*);

atan2 calculates the inverse tangent of the quotient of its arguments *num* and *den*. These may be any real number except zero.

atan2 returns the result, which is in the range of from -π to π. The sign of the return value is drawn from the signs of both arguments.

Cross-references

Standard, §4.5.2.4
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan, cos, mathematics, sin, tan

Notes

atan2 is provided in addition to **atan**, to compute arc tangents for numbers that yield very large results.

atexit() — General utility (libc)

Register a function to be performed at exit
#include <stdlib.h>
int atexit(void (*function**)(void));**

atexit registers a function to be executed when the program exits. *function* points to the function to be executed. The registered function returns nothing. **atexit** provides a way to perform additional clean-up operations before a program terminates.

The functions that **atexit** registers are executed when the program exits normally, i.e., when the function **exit** is called or when **main** returns. The functions registered by **atexit** can perform clean-up is needed, beyond what is ordinarily performed when a program exits.

atexit returns zero if *function* could be registered, and nonzero if it could not.

Example

This example sets one function that displays messages when a program exits, and another that waits for the user to press a key before terminating.

```
#include <stdlib.h>
#include <stdio.h>

void
lastgasp(void)
{
      perror("Type return to continue");
}
```

```
void
get1(void)
{
      getchar();
}

main(void)
{
      /* set up get1() as last exit routine */
      atexit(get1);
      /* set up lastgasp() as exit routine */
      atexit(lastgasp);

      /* exit, which invokes exit routines */
      exit(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.4.2
The C Programming Language, ed. 2, p. 253

See Also

exit, general utility

Notes

atexit must be able to register at least 32 functions.

Functions registered by **atexit** are executed when **exit** is called. They are executed in *reverse* order of registration.

atof() — General utility (libc)

Convert string to floating-point number
#include <stdlib.h>
double atof(const char ****string*);**

atof converts the string pointed to by *string* into a double-precision floating point number, and returns the number it has built. It is equivalent to the call

```
strtod(string, (char **)NULL);
```

string must point to the text representation of a floating-point number. It can contain a leading sign, any number of decimal digits, and a decimal point. It can be terminated with an exponent, which consists of the letters 'e' or 'E' followed by an optional leading sign and any number of decimal digits. For example,

```
1.23
123e-2
123E-2
```

are strings that can be converted by **atof**.

atof ignores leading blanks and tabs; it stops scanning when it encounters any unrecognized character.

Cross-references

Standard, §4.10.1.1
The C Programming Language, ed. 2, p. 251

LEXICON

See Also

atoi, atol, general utility, strtod, strtol, strtoul

Notes

The character that **atof** recognizes as representing the decimal point depends upon the program's locale, as set by the function **setlocale**. See **localization** for more information.

The functionality of **atof** has largely been subsumed by the function **strtod**, but the Standard includes it because it is used so widely in existing code.

atoi() — General utility (libc)

Convert string to integer
#include <stdlib.h>
int atoi(const char ******string***);**

atoi converts the string pointed to by *string* into an integer. It is equivalent to the call

```
(int)strtol(string, (char **)NULL, 10);
```

The string pointed to by *string* may contain a leading sign and any number of numerals. **atoi** ignores all leading white space. It stops scanning when it encounters any non-numeral other than the leading sign character and returns the **int** it has built.

Cross-references

Standard, §4.10.1.2
The C Programming Language, ed. 2, p. 251

See Also

atof, atol, general utilities, strtod, strtol, strtoul

Notes

The functionality of **atoi** has largely been subsumed by the function **strtol**, but the Standard includes it because it is used so widely in existing code.

atol() — General utility (libc)

Convert string to long integer
#include <stdlib.h>
long atol(const char ******string***);**

atol converts the string pointed to by *string* to a **long**. It is equivalent to the call

```
strtol(string, (char **)NULL, 10);
```

The string pointed to by *string* may contain a leading sign and any number of numerals. **atol** ignores all leading white space. It stops scanning when it encounters any non-numeral other than the leading sign and returns the **long** it has built.

Cross-references

Standard, §4.10.1.3
The C Programming Language, ed. 2, p. 251

See Also

atof, atol, general utilities, strtod, strtol, strtoul

Notes

The functionality of **atol** has largely been subsumed by the function **strtol**, but the Standard includes it because it is used so widely in existing code.

auto — C keyword

Automatic storage duration
auto *type identifier*

The storage-class specifier **auto** declares that *identifier* has automatic storage duration.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, p. 210

See Also

storage-class specifiers, storage duration

aux — Operating system device

Logical device for serial port

MS-DOS gives names to its logical devices. **Let's C** uses these names to access these devices via MS-DOS.

aux is the logical device for the the serial port auxiliary device.

Example

The following example opens the auxiliary port and sends it the string **hello, world**.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      FILE *fp, *fopen();
      if ((fp = fopen("aux", "w")) != NULL) {
            printf("aux enabled\n");
            fprintf(fp, "hello, world.\n");
      }
      else printf("aux: cannot open.\n");
      return EXIT_SUCCESS;
}
```

See Also

com1, con, crts, lpt1, nul, operating system devices

B

behavior — Definition

The term *behavior* refers to the way an implementation reacts to a given construct. When a construct conforms to the descriptions within the Standard, then its behavior should be predictable from the Standard's descriptions alone. When a construct does not conform to the descriptions within the Standard, then one of the following four types of abnormal behavior results:

Unspecified behavior
: This is behavior produced by a correct construct for which the Standard supplies no description. An example is the order in which a program evaluates the arguments to a function.

Undefined behavior
: This is behavior produced by an erroneous construct for which the Standard supplies no description. An example of a construct that generates undefined behavior is attempting to divide by zero.

: The Standard does not mandate how a conforming implementation reacts when it detects a construct that will produce undefined behavior: it may pass over it in silence, with unpredictable (and usually unwelcome) results; generate a diagnostic message and continue to translate or execute; or stop translation or execution and produce a diagnostic message.

: A portable program, however, should not depend upon undefined behavior performing in any predictable way. Undefined behavior is precisely that: undefined. Whatever happens, happens — from printing an error message to reformatting your hard disk.

Implementation-defined behavior
: This is behavior produced by a correct construct that is specific to a given implementation. An example is the number of **register** objects that can actually be loaded into machine registers. The Standard requires that the implementation document all such behaviors.

Locale-specific behavior
: This is behavior that depends upon the program's locale. An example is the character that the function **atof** recognizes as marking a decimal point. The Standard requires that an implementation document all such behaviors.

Cross-reference

Standard, §1.6

See Also

compliance, Definitions

Notes

For a program to be maximally portable, it should not rely on any of the above deviants of behavior.

BIOS — Definition

BIOS is an acronym for *basic input/output system*. In most machines, the BIOS consists of a group of routines carried in the read-only memory (ROM). These routines contain basic instructions for accessing the various aspects of the hardware. MS-DOS uses these routines to help protect itself from the peculiarities of the hardware on which it is running.

See Also

Definitions, STDIO

bios.h — Header

Outline ROM BIOS data area

bios.h is a header file to be used with programs that directly access the IBM PC's BIOS data area. It declares a structure that defines the entire BIOS data area, for examination and alteration.

The sample program **biosdata.c**, which is included with **Let's C**, uses **bios.h** to take a "snapshot" of the BIOS data area and print a summary of it.

See Also

BIOS data area, header, peek, poke

bit — Definition

The term *bit* is an abbreviation for *binary digit*. It is the element of storage that can hold either of exactly two values. A contiguous sequence of bits forms a byte. A byte consists of at least eight bits. The macro **CHAR_BIT** specifies the number of bits that constitute a byte for the execution environment.

On most machines a bit cannot be addressed directly; a byte is the smallest unit of storage that can be addressed.

Cross-reference

Standard, §1.6

See Also

bit-field, bitwise operations, byte, Definitions

bit-fields — Definition

A *bit-field* is a member of a structure or **union** that is defined to be a cluster of bits. It provides a way to represent data compactly. For example, in the following structure

```
struct example {
      int member1;
      long member2;
      unsigned int member3 :5;
}
```

member3 is declared to be a bit-field that consists of five bits. A colon ':' precedes the integral constant that indicates the *width*, or the number of bits in the bit-field. Also, the bit-field declarator must include a type, which must be one of **int**, **signed int**, or **unsigned int**. If a bit-field is declared to be in type **int**, the implementation defines whether the highest bit is used to hold the bit-field's sign.

The Standard states, "An implementation may allocate any addressable storage unit large enough to hold a bit-field." This suggests that if a bit-field is defined as holding more bits than are normally held by an **int**, then the implementation may place the bit-field into a larger data object, such as a **long**.

If two bit-fields are declared side-by-side and together are small enough to fit into an **int**, then they must be packed together. However, if together they are too large to fit into an **int**, then the implementation determines whether they are in separate objects or if the second bit-field is partly within the object that holds the first and partly within a second object.

The implementation also defines where the bit-field resides within its object — whether it is built from the low-order bit up, or from the high-order bit down. For example, consider an implementation in which an **int** has 16 bits. If a five-bit bit-field is declared to be part of an **int**, and that bit-field is initialized to all ones, then the **int** may appear like this under one

implementation:

```
0000 0000 0001 1111     /* low-order bits set */
```

and like this under another:

```
1111 1000 0000 0000     /* high-order bits set */
```

A bit-field that is not given a name may not be accessed. Such an object is useful as "padding" within an object so that it conforms to a template designed elsewhere.

A bit-field that is unnamed and has a length of zero can be used to force adjacent bit-fields into separate objects. For example, in the following structure

```
struct example {
      int member1;
      int member2 :5;
      int :0;
      int member3 :5;
};
```

the zero-length bit-field forces **member2** and **member3** to be written into separate objects, regardless of the default behavior of the implementation.

Finally, it is not allowed to take the address of a bit-field.

Cross-references

Standard, §3.5.2.1
The C Programming Language, ed. 2, pp

See Also

bit, bit map, byte, Definitions

Notes

Because bit-fields have many implementation-specific properties, they are not considered to be highly portable. Bit-fields use minimal amounts of storage, but the amount of computation needed to manipulate and access them may negate this benefit. Bit-fields must be kept in integral-sized objects because many machines cannot access a quantity of storage smaller than a "word" (a word is generally used to store an **int**).

bit map — Definition

A **bit map** is a string of bits in which each bit has a symbolic, rather than numeric, value.

See Also

bit, byte, Definitions

Notes

C permits the manipulation of bits within a byte through the use of bit field routines. These generate code rather than calls to routines. Bit fields are generally less efficient than masking because they always generate masking and shifting.

block — Definition

A *block* is a set of statements that forms one syntactic unit. It can have its own declarations and initializations.

In C terminology, a block is marked off by braces '{ }'. Block-scoped variables are visible only in the block in which they are declared.

Cross-references

Standard, §3.6.2
The C Programming Language, ed. 2, p. 55

See Also

auto, compound statement, Definitions, scope

Notes

Another term for "block" is *compound statement*.

break — C keyword

Exit unconditionally from loop or switch
break;

break is a statement that causes the program to exit immediately from the smallest enclosing **switch**, **while**, **for**, or **do** statement.

Example

For an example of this statement, see **printf**.

Cross-references

Standard, §3.6.6.3
The C Programming Language, ed. 2, p. 64

See Also

C keywords, continue, goto, statements, return

bsearch() — General utility (libc)

Search an array
#include <stdlib.h>
void *bsearch(**const void** **item*, **const void** **array*, **size_t** *number*,
size_t *size*, **int** (**comparison*)(**const void** **arg1*, **const char** **arg2*));

bsearch searches a sorted array for a given item.

item points to the object sought. *array* points to the base of the array; it has *number* elements, each of which is *size* bytes long. Its elements must be sorted into ascending order before it is searched by **bsearch**.

comparison points to the function that compares *item* with an element of *array*. *comparison* must return zero if its arguments match, a number greater than zero if the element pointed to by *arg1* is numerically greater than the element pointed to by *arg2*, and a number less than zero if the element pointed to by *arg1* is numerically less than the element pointed to by *arg2*.

bsearch returns a pointer to the array element that matches *item*. If no element matches *item*, then **bsearch** returns NULL. If more than one element within *array* matches *item*, which element is matched is unspecified.

Example

This example uses **bsearch** to translate English into "bureaucrat-ese".

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

LEXICON

```
struct syntab {
      char *english, *bureaucratic;
} cdtab[] = {
/* The left column is in alphabetical order */

      "affect",        "impact",
      "after",         "subsequent to",
      "building",      "physical facility",
      "call",          "refer to as",
      "do",            "implement",

      "false",         "inoperative",
      "finish",        "finalize",
      "first",         "initial",
      "full",          "in-depth",
      "help",          "facilitate",

      "lie",           "inoperative statement",
      "order",         "prioritize",
      "talk",          "interpersonal communication",
      "then",          "at that point in time",
      "use",           "utilize"
};

int
comparator(key, item)
char *key;
struct syntab *item;
{
      return(strcmp(key, item->english));
}

main(void)
{
      struct syntab *ans;
      char buf[80];

      for(;;) {
            printf("Enter an English word: ");
            fflush(stdout);

            if(gets(buf) || !strcmp(buf, "quit") == NULL)
                  break;

            if((ans = bsearch(buf, (void *)cdtab,
                        sizeof(cdtab)/ sizeof(struct syntab),
                        sizeof(struct syntab),
                        comparator)) == NULL)
                  printf("%s not found\n");

            else
                  printf("Don't say \"%s\"; say \"%s\"!\n",
                        ans->english, ans->bureaucratic);
      }

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.5.1
The C Programming Language, ed. 2, p. 253

See Also

qsort, searching-sorting

byte — Definition

A *byte* is a contiguous set of at least eight bits. It is the unit of storage that is large enough to hold each character within the basic C character set. It is also the smallest unit of storage that a C program can address.

The least significant bit is called the *low-order bit*, and the most significant bit is the *high-order bit*.

In terms of C programming, a byte is synonymous with the data type **char**: a **char** is defined to be equal to one byte's worth of storage. The macro **CHAR_BIT** gives the number of bits in a byte for the execution environment.

Cross-reference

Standard, §1.6

See Also

bit, char, Definitions

byte ordering — Technical information

Describe order of bytes

Byte ordering is the order in which a given machine stores successive bytes of a multibyte data item. Different machines order bytes differently.

The following example displays a few simple examples of byte ordering:

```
#include <stddef.h>
#include <stdio.h>
#include <stddef.h>

main(void)
{
      union
      {
            char b[4];
            int i[2];
            long l;
      } u;
      u.l = 0x12345678L;

      printf("%x %x %x %x\n",
            u.b[0], u.b[1], u.b[2], u.b[3]);
      printf("%x %x\n", u.i[0], u.i[1]);
      printf("%lx\n", u.l);
      return(EXIT_SUCCESS);
}
```

When run on the 68000 or the Z8000, the program gives the following results:

```
12 34 56 78
1234 5678
12345678
```

As you can see, the order of bytes and words from low to high memory is the same as is represented on the screen.

When run on a PDP-11, however, the program gives these results:

LEXICON

```
34 12 78 56
1234 5678
12345678
```

As you can see, the PDP-11 inverts the order of words in memory.

Finally, when the program is run on the i8086 you see these results:

```
78 56 34 12
5678 1234
12345678
```

The i8086 inverts both words and long words.

See Also

Language, technical information

C

cabs() — Extended function (libm)

Complex absolute value function
#include <xmath.h>
double cabs(struct { double *r*, *i*; } *z*);

cabs computes the absolute value, or modulus, of its complex argument *z*. The absolute value of a complex number is the length of the hypotenuse of a right triangle whose sides are given by the real part *r* and the imaginary part *i*. The result is the square root of the sum of the squares of the parts.

See Also

extended mathematics, hypot

Notes

To conform to the ANSI Standard, **cabs** has been moved from the header **math.h** to the header **xmath.h**. This may require that some code be altered.

cabs is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

calloc() — General utility (libc)

Allocate and clear dynamic memory
#include <stdlib.h>
void *calloc(size_t *count*, **size_t** *size*);

calloc allocates a portion of memory large enough to hold *count* items, each of which is *size* bytes long. It then initializes every byte within the portion to zero.

calloc returns a pointer to the portion allocated. The pointer is aligned for any type of object. If it cannot allocate the amount of memory requested, it returns NULL.

Example

For an example of this function, see **stdarg**.

Cross-references

Standard, §4.10.3.1
The C Programming Language, ed. 2, p. 167

See Also

alignment, free, general utilities, malloc, realloc

Notes

If *count* or *size* is equal to zero, then the behavior of **calloc** is implementation defined: **calloc** returns either NULL or a unique pointer. This is a quiet change that may silently break some existing code.

case — C keyword

Mark entry in switch table
case *expression*:

case is a label that introduces an entry within the body of a **switch** statement. The value of the **switch** statement's conditional expression is compared with the value of every **case** label's expression. When the two match, then the program jumps to the point marked by that **case** label and execution continues from there. Execution continues until a **break** statement is encountered.

Each **case** label must mark an expression whose value differs from those of every other **case** label for that **switch** statement. See **switch** for more information.

Example

For an example, see **printf**.

Cross-references

Standard, §3.6.1
The C Programming Language, ed. 2, p. 58

See Also

break, C keywords, default, statements, switch

Notes

Every conforming implementation must be able to accept at least 257 **case** labels within a **switch** statement.

cc — Command

Compiler controller
cc [*options***]** *file . . .*

cc is the program that controls compilation. It guides files of source and object code through each phase of compilation and linking. **cc** has many options to assist in the compilation of C programs; in essence, however, all you need to do to produce an executable file from your C program is type **cc** followed by the name of the file or files that hold your program. It checks whether the file names you give it are reasonable, selects the right phase for each file, and performs other tasks that ease the compilation of your programs.

File Names

cc assumes that each *file* name that ends in **.c** or **.h** is a C program and passes it to the C compiler for compilation.

cc assumes that each *file* argument that ends in **.s** is in Mark Williams assembly language and processes it with the assembler **as**; and that every file with the suffix **.asm** is written in Microsoft macro assembly language, and attempts to assemble it with the macro assembler **MASM**.

cc assumes that all files with the suffix **.m** are assembly-language files that also use C preprocessor instructions. **cc** will pass these files to the C preprocessor **cpp**, and pass its output to the assembler **as**. This hybrid allows you to write assembly-language programs that are model-independent. For an example of a **.m** file, see the Lexicon entry for **as**, the assembler. For more information on building **.m** files, see the entry for **larges.h**.

cc also passes all files with the suffixes **.obj** or **.lib** unchanged to the linker MS-LINK.

How cc Works

cc normally works as follows: First, it compiles or assembles the source files, naming the resulting object files by replacing the **.c**, **.s**, **.m**, or **.asm** suffixes with the suffix **.obj**. Then, it links the object files with the C runtime startup routine and the standard C library, and leaves the result in file *file***.exe**. If only one object file is created during compilation, it is deleted after linking; however, if more than one object file is created, or if an object file of the same name existed before you began to compile, then the object file or files are not deleted.

Arguments and Wildcards

The option **-na** (for "no arguments") tells **Let's C** that a program does not take command-line arguments. This option may be used with or without the **-ns** option, which suppresses the linking

of STDIO into your program.

The option **-w** (for "wildcard") tells **Let's C** to include code in your program that will allow it to expand the wildcard characters '?' and '*' in command-line arguments. For example, if program **example.exe** is *not* built with the **-w** option, the command

```
example *.c
```

results in an argument count of two, and an argument list that contains two non-NULL members. If **example.exe** is built *with* the **-w** option, **Let's C** will include code so that your program will automatically expand the wildcard argument ***.c**. The argument count and argument list are altered to reflect the number of such files and their names, respectively.

If a program defines a global array **char _cmdname[]** that gives the name of the command, the **-w** option fills in **argv[0]** with the command name and looks for environmental variables of the form *<name>***HEAD** and *<name>***TAIL**. If found, these are added to **argv[]** before and after command-line arguments, respectively. This option limits your program to 256 arguments at any one time. If you happen to need to use more than 256 arguments, use the program **msdoscvt**, which is presented as an example in the entry for **exargs**.

For example, the **wc** command is built with the **-w** option and defines

```
_cmdname = "wc";
```

If the current directory contains files **a.c** and **b.c**, and environmental variable **WCHEAD** is set to **-l**, the command

```
wc *.c
```

has the same effect as the command

```
wc -l a.c b.c
```

that is, it counts the lines in **a.c** and **b.c**.

The arguments to **main** are defined as

```
main(argc, argv)
int argc; char *argv[];
```

On some systems, a third argument is available:

```
main(argc, argv, envp)
int argc; char *argv[], *envp[];
```

The **envp** argument is a NULL-terminated array of pointers to environmental variables, each of the form *var=value*. If a program is compiled without the **-w** option, **Let's C** passes an empty list as **envp**. If a program is compiled with the **-w** option, **Let's C** passes an **envp** that contains MS-DOS environmental variables.

Options

The following lists all of **cc**'s command-line options. **cc** passes some options through to the linker MS-LINK unchanged, and correctly interprets to it the options **-o**, **-u**, **-y/**, **-yf**, **-ym**, **-yn**, **-ys**, and **-yu**.

A number of the options are esoteric and normally are not used when compiling a C program. The following are the most commonly used options:

LEXICON

-A	invoke MicroEMACS when errors occur
-f	include floating-point **printf**
-l*name*	pass library **lib***name***.lib** to linker
-o *name*	call executable file *name*
-V	print details of compiler's actions
-VASM	generate assembly-language output

-A MicroEMACS option. If an error occurs during compilation, **cc** automatically invokes the MicroEMACS screen editor. The error or errors are displayed in one window and the source code file in the other, with the cursor set to the line number indicated by the first error message. Typing **<ctrl-X>>** moves to the next error, **<ctrl-X><** moves to the previous error. To recompile, close the edited file with **<ctrl-Z>**. Compilation will continue either until the program compiles without error, or until you exit from the editor by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

-c Compile option. Suppress linking and the removal of the object files.

-cc2l Use a LARGE-model version of the code generator **cc2**. This allows the creation of extremely large programs, but runs more slowly than the default **cc2**, which is in SMALL model.

*-D***name**[=**value**]
Define *name* to the preprocessor, as if set by a **#define** directive. If *value* is present, it is used to initialize the definition.

-E Expand option. Run the C preprocessor **cpp** and write its output onto the standard output.

-f Floating point option. Include library routines that perform floating-point arithmetic. Because the floating-point routines require approximately five kilobytes of memory, the standard C library does not include them; the **-f** option tells the compiler to include them. If a program is compiled without the **-f** option but attempts to print a floating point number during execution by using the **e**, **f**, or **g** format specifications to **printf**, the message

```
You must compile with -f option for floating point
```

will be printed and the program will exit.

-I*directory*
Include option. Specify the directory the preprocessor should search for files given in **#include** directives, using the following criteria: If the **#include** statement reads

```
#include "file.h"
```

cc searches for **file.h** first in the source directory, then in the directory named in the **-I***directory* option, and finally in the system's default directories. If the **#include** statement reads

```
#include <file.h>
```

cc searches for **file.h** first in the directory named in the **-I***directory* option, and then in the system's default directories. Multiple **-I***directory* options are executed in their order of appearance.

-K Keep option. Do not erase the intermediate files generated during compilation. Temporary files will be written into the current directory. The **-K** option takes precedence over the **-xt** option: when **-K** is set, the temporary files are always written into the directory in which the source code is kept.

-l *name*
library option. Pass the name of a library to the linker. **cc** expands **-l***name* into **lib***name***.a**.

-m Mini-**make** option: Compile file of source code only if it has been changed since its identically changed object file was last compiled.

-na No arguments option. The compiled program does not use **argc** or **argv**. See *Arguments and wildcards*, above, for more information.

-ns Do not link in **stdio**. If the standard I/O library is not needed, the **-ns** option produces much smaller object modules.

-o *name*
Output option. Rename the executable file from the default *file***.exe** to *name*.

-U *name*
Undefine symbol *name*. Use this option to undefine symbols that the preprocessor defines implicitly, such as the name of the native system or machine.

-V Verbose option. **cc** prints onto the standard output a step-by-step description of each action it takes.

V*string*
Variant option. *Toggle* (i.e., turn on or off) the variant *string* during the compilation. Options marked **Strict:** generate messages that warn of the conditions in question. If you name an option once in the **CCHEAD** environmental variable and again on the **cc** command line, these two togglings will cancel each other out. **cc** recognizes the following variants:

-V80186 Output code that uses the instructions native to the Intel i80186 and i80286 microprocessors. This switch also works with the assembler **as**: assembly-language programs that contain i80186/286 instructions will be assembled correctly when the assembler is invoked using this option. Programs compiled with this option cannot be run on an IBM PC or strictly compatible machines, but will take full advantage of the instruction set of the IBM AT and its compatibles. The code will also execute correctly on the NEC V20 and V30 processors. Default is **off**.

-VALIEN Enable the **alien** keyword. Under **Let's C**, the **alien** keyword allows direct calls of PL/M, Pascal, and FORTRAN functions and procedures. These differ from C functions in the following ways: (1) C pushes arguments from right to left; the other languages push from left to right. (2) C arguments are popped by the calling function, whereas under the other languages arguments are popped by the called function. (3) **Let's C** appends an underbar character to the end of every function name, whereas the other languages do not. Default is **off**.

-VASM Output assembly-language code. It can be used with the **-VLINES** option, described below, to generate a line-numbered file of assembly language. Default is **off**.

-VCSD Generate debugging information for **csd**, the Mark Williams C Source Debugger.

-VFLOAT Include floating point **printf** routines. Same as **-f** option, above.

-VLARGE LARGE-model output. Default is **off**.

-VLINES Generate line number information. Can be used with the option **-S**, described above to generate assembly language output that uses line numbers. Default is **off**.

LEXICON

-VNDP Generate i8087 floating-point code. The code generated with this option will run only on machines that have an i8087 mathematics co-processor. If this option is *not* used, **Let's C** automatically uses libraries that sense the presence of the i8087: if an i8087 is present, floating point routines will be run on it; but if one is not present, they will be emulated in software. Default is **off**.

-VOPT Turn on optimization. Default is **off**.

-VPSTR Put strings into the shared segment, if possible. Used to generate ROMable code. Default is **off**.

-VQUIET Suppress all messages. Default is **off**.

-VSBOOK Strict: note deviations from *The C Programming Language*, ed. 2. Default is **off**.

-VSLCON Strict: **int** constant promoted to **long** because value is too big. Default is **on**.

-VSMALL Generate SMALL-model output. Default is **on**.

-VSMEMB Strict: check use of structure/union members for adherence to standard rules of C. Default is **on**.

-VSNREG Strict: register declaration reduced to auto. Default is **on**.

-VSPVAL Strict: pointer value truncated. Default is **off**.

-VSRTVC Strict: risky types in truth contexts. Default is **off**.

-VSTAT Give statistics on optimization.

-VSTRICT Turn on all strict checking. Default is **on**.

-VSUREG Strict: note unused registers. Default is **off**.

-VSUVAR Strict: note unused variables. Default is **on**.

-V3GRAPH
Translate ANSI trigraphs. Default is **off**.

-w Wildcards option: the compiled program can take wildcards in its command line. See *Arguments and wildcards*, above, for more information.

-x<*key***><***directory***>**
Use the given *directory* as the location for one of the following: for compiler files if *key* is **c**; libraries if *key* is **l**; output files if *key* is **o**; or temporary files if *key* is **t**.

-y/*switch*
Pass *switch* directly to MS-LINK.

-yf Force MS-LINK to create a linker command file. For more information on what a linker command file is, see the Lexicon entry for **MS-LINK**.

-ym Force MS-LINK to create a map file. For more information on what a map file is, see the Lexicon entry for **MS-LINK**.

-yn Reset the MS-LINK segments to 1,024, using the form **/segments=1024** required by MS-LINK versions 3.02 and later.

-ys*number*
Force MS-LINK to set the stack size to *number*, where *number* gives the number of bytes of stack required, in decimal figures.

LEXICON

-yu*name*
Undefine *name* for MS-LINK.

-Z Pause between passes and prompt for disk change. Used with the compiler using single-sided disks.

See Also

as, cc0, cc1, cc2, cc3, commands, cpp, ld

cc0 — Definition

cc0 is the **Let's C** *preprocessor* and *parser*. It performs all preprocessing tasks, and parses C programs using the method of recursive descent. It then translates the program into a logical-tree format.

See Also

cc, cc1, cc2, cc3, cpp, Definitions, preprocessing

cc1 — Definition

cc1 is the **Let's C** code generator. This phase generates code from the trees created by the parser, **cc0**. Code generation is table driven, with entries for each operator and addressing mode.

See Also

cc, cc0, cc2, cc3, cpp, Definitions

cc2 — Definition

cc2 is the optimizer/object generator phase of **Let's C**. It optimizes the code generated by **cc1**, and writes the object code.

Let's C uses multiple optimization algorithms. One optimizes jump sequences: it eliminates common code, optimizes span-dependent jumps, and removes jumps to jumps. The other function scans the generated code repeatedly to eliminate unnecessary instructions.

The **cc** option **-cc2l** uses a LARGE-model version of **cc2**. This allows you to create extremely large programs, but runs more slowly than the default version of **cc2**, which is in SMALL model.

See Also

cc, cc0, cc1, cc3, cpp, Definitions

cc3 — Definition

cc3 is the output phase of **Let's C** that writes a file of assembly language rather than a relocatable object module. This phase is optional. It allows you to examine the code generated by the compiler. To produce an assembly-language output of a C program, use the **-VASM** option on the **cc** command line. For example,

```
cc -VASM foo.c
```

tells **cc** to produce a file of assembly language called **foo.s**, instead of an object module.

See Also

cc, cc0, cc1, cc2, cpp, Definitions

CCTAIL — Environmental variable

Variables at end of compilation command

CCTAIL is an environmental variable that is read by the **cc** command. When you issue a **cc** command, **cc** reads **CCTAIL** and appends it to the end of the list of arguments you have given **cc**.

You should set **CCTAIL** in **autoexec.bat** to add options routinely to your **cc** commands. For example, adding the command

```
set CCTAIL=-lm
```

to **autoexec.bat** ensures that the mathematics library **libm.lib** is always linked into your C programs. Thus, typing the command

```
cc foo.c
```

will have the same effect as typing

```
cc foo.c -lm
```

See Also

cc, CCHEAD, environmental variable

ceil() — Mathematics (libm)

Integral ceiling
#include <math.h>
double ceil(double *z***);**

The function **ceil** returns the "ceiling" of a function, or the smallest integer less than *z*. For example, the ceiling of 23.2 is 23, and the ceiling of -23.2 is -23.

ceil returns the value expressed as a **double**.

Cross-references

Standard, §4.5.6.1
The C Programming Language, ed. 2, p. 251

See Also

fabs, floor, fmod, mathematics

char — C keyword

The data type **char** is the smallest addressable unit of data. It consists of one byte of storage, and it can encode all of the characters that can be used to write a C program. **sizeof(char)** returns one by definition, with all other data types defined as multiples thereof.

A **char** may be either signed or unsigned; this is up to the implementation. **Let's C** uses a signed **char** by default. If a **char** holds any of the characters that make up the C character set, then it is positive. ANSI C allows the corresponding types **signed char** and **unsigned char**. Programmers can create signed and unsigned versions of **char** where needed.

The range of values that can be encoded within a **char** are set by the macros **CHAR_MIN** and **CHAR_MAX**. These are defined in the header **limits.h**. The minimum values of these macros depend upon whether the implementation sign-extends a **char** when it is used in an expression. If the implementation does sign extend, then **CHAR_MIN** is equal to **SCHAR_MIN** (at least -127) and **CHAR_MAX** is equal to **SCHAR_MAX** (at least +127). If it does not sign extend, however, **CHAR_MIN** is equal to zero and **CHAR_MAX** is equal to **UCHAR_MAX** (at least +255).

Cross-references

Standard, §3.1.2.5
The C Programming Language, ed. 2, p. 211

See Also

signed char, types, unsigned char

character constant — Definition

A *character constant* is a constant that encodes a character or escape sequence. A character constant consists of one or more characters or escape sequences that are enclosed within apostrophes ´. To include a literal apostrophe within a character constant, use the escape sequence **\'**.

A character is regarded as having type **char** as it is read, and it yields an object with type **int**. If a character constant contains one character or escape sequence, then the numeric value of that character is written into an **int**-length object. For example, under an implementation that uses ASCII, the character constant **'a'** yields an **int**-length object with the value of 0x61. If a character constant contains more than one character or escape sequence, the result is implementation-defined.

Because the constant being read is regarded as having type **char**, the value of a character constant can change from implementation to implementation, depending upon whether the implementation uses a signed or unsigned **char** by default. For example, in an environment in which a **char** has eight bits and uses two's-complement arithmetic, the character constant **'\xFF'** will yield an **int** with a value of either -1 or +255, depending upon whether a **char** is, respectively, signed or unsigned by default. **Let's C** uses signed **char**s by default.

A *wide-character constant* is a character constant that is formed of a wide character instead of an ordinary, one-byte character. It is marked by the prefix 'L'. For example, in the following

```
L'm';
```

stores the numeric value of 'm' in the form of a wide character.

Example

For an example of using character constants in a program, see **putchar**.

Cross-references

Standard, §3.1.3.4
The C Programming Language, ed. 2, p. 193

See Also

constants, escape sequences

Notes

Although octal escape sequences are limited to three octal digits, hexadecimal escape sequences can be arbitrary length. However, when the value of a hexadecimal escape sequence exceeds that which can be represented in an **int**, behavior is defined by the implementation.

character display semantics — Definition

The Standard describes the semantics by which characters are displayed on an output device. The *active position* is where the output device will print the next character produced by the function **fputc**. On a video terminal, it usually is marked by a cursor. The locale defines the direction of printing, whether from left to right, from right to left, or from top to bottom.

The following escape sequences can be embedded within a string literal or character constant to

affect the behavior of an output device:

\a Generate an alert signal. The alert may take the form of ringing a bell or printing a visual signal on a screen.

\b Backspace: move the active position back one position. If the active position is already at the beginning of the line, the behavior is undefined.

\f Form feed: move the active position to the beginning of the next page. On a hard-copy printer, it feeds a fresh sheet of paper. On a video terminal, it may take the form of clearing the screen and moving the cursor to the "home" position.

\n Newline: move the active position to the beginning of the next line.

\r Return: move the active position to the beginning of the current line.

\t Horizontal tab: move the active position to the beginning of the next horizontal tabulation field. If the active position is already at or past the last horizontal tabulation field on the current line, the behavior is undefined.

\v Vertical tab: move the active position to the beginning of the next vertical tabulation field. If the active position is already at or past the last vertical tabulation field, the behavior is undefined.

Every implementation must define each of these escape sequences as being a unique value that can be stored in one **char** object.

Cross-reference

Standard, §2.2.2

See Also

Environment, escape sequence, trigraph sequences

character handling — Overview

#include <ctype.h>

The Standard's repertoire of library functions includes 13 that test or alter individual characters, as follows:

Character testing

isalnum	Check if a character is a numeral or letter
isalpha	Check if a character is a letter
iscntrl	Check if a character is a control character
isdigit	Check if a character is a numeral
isgraph	Check if a character is printable
islower	Check if a character is a lower-case letter
isprint	Check if a character is printable
ispunct	Check if a character is a punctuation mark
isspace	Check if a character is white space
isupper	Check if a character is a upper-case letter
isxdigit	Check if a character is a hexadecimal numeral

Case mapping

tolower	Convert character to lower case
toupper	Convert character to upper case

All are declared in the header **ctype.h**.

The operation of all character-handling functions (with the exception of **isdigit** and **isxdigit**) is modified by the program's locale, as set by the function **setlocale**. This allows these function to test and modify characters using a locale-specific character set. The calls

```
setlocale(LC_CTYPE, locale);
```

or

```
setlocale(LC_ALL, locale);
```

force these functions to use the locale-specific character set. See **localization** for more information.

Cross-references

Standard, §4.3
The C Programming Language, ed. 2, p. 248

See Also

character, ctype.h, extended character handling, Library

Notes

Although these functions are described as "character handling," they are defined as taking an argument of type **int** to allow them to accept the special value of **EOF** and locale-specific character sets.

clearerr() — STDIO (stdio.h)

Clear a stream's error indicator
#include <stdio.h>
void clearerr(FILE *_fp_**);**

When a file is manipulated, a condition may occur that would cause trouble should the program continue. This could be an error (e.g., a read error), or the program may have read to the end of the file. Most environments use two indicators to signal that such a condition has occurred: the *error indicator* and the *end-of-file* indicator.

When an error occurs, the error indicator is set to a value that indicates what error occurred. The end-of-file indicator is set when the end of a file is read. By checking these indicators, a program can see if all is going well. A file may not be manipulated further until both indicators have been reset to their normal values.

clearerr resets to normal the error indicator and the end-of-file indicator for the stream pointed to by *fp*.

Cross-references

Standard, §4.9.10.1
The C Programming Language, ed. 2, p. 248

See Also

feof, ferror, perror, STDIO

Notes

The indicators are cleared when a file is opened or when the file-position indicator is reset by the function **rewind**. Successful calls to **fseek**, **fsetpos**, or **ungetc** clear the end-of-file indicator.

CLK_TCK — Manifest constant

#include <time.h>
CLK_TCK is a manifest constant that is defined in the header **time.h**. It represents the number of "ticks" in a second. A "tick" is the unit of time measured by the function **clock**.

clock returns the type **clock_t**. To determine how many seconds a program required to run to the given point, divide the value returned by **clock** by the value of **CLK_TCK**.

Example

For an example of using this macro in a program, see **clock**.

Cross-references

Standard, §4.12.1
The C Programming Language, ed. 2, p. 255

See Also

clock, clock_t, date and time

clock() —

Get processor time used
#include <time.h>
clock_t clock(void);

clock calculates and returns the amount of processor time a program has taken to execute to the current point. Execution time is calculated from the time the program was invoked. This, in turn, is set as a point from the beginning of an era that is defined by the implementation. For example, under the COHERENT operating system, time is recorded as the number of milliseconds since January 1, 1970, 0h00m00s UTC.

The value **clock** returns is of type **clock_t**. This type is defined in the header **time.h**. The Standard defines it merely as being an arithmetic type capable of representing time. If **clock** cannot determine execution time, it returns -1 cast to **clock_t**.

To calculate the execution time in seconds, divide the value returned by **clock** by the value of the macro **CLK_TCK**, which is defined in the header **time.h**.

Example

This example measures the number of times a **for** loop can run in one second on your system. This is approximate because **CLK_TCK** can be a real number, and because the program probably will not start at an exact tick boundary.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void)
{
      clock_t finish;
      long i;

      /* finish = about 1 second from now */
      finish = clock() + CLK_TCK;
      for(i = 0; finish > clock(); i++)
            ;
```

```
      printf("The for() loop ran %ld times in one second.\n", i);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.2.1
The C Programming Language, ed. 2, p. 255

See Also

CLK_TCK, clock_t, date and time, difftime, mktime

clock_t — Type

System time
#include <time.h>

clock_t is a data type that is defined in the header **time.h**. It is an arithmetic type, and is the type returned by the function **clock**.

The unit that **clock_t** holds is implementation-defined. The manifest constant **CLK_TCK** expands to a number that expresses how of many of these units constitute one second of real time.

Example

For an example of using this type in a program, see **clock**.

Cross-references

Standard, §4.12.1
The C Programming Language, ed. 2, p. 255

See Also

CLK_TCK, clock, date and time, time_t

close() — Extended function (libc)

Close a file
short close(short *fd***);**

close closes the file identified by the file descriptor *fd*, which was returned by **creat**, **dup**, or **open**. **close** frees the associated file descriptor.

close returns -1 if an error occurs, such as its being handed a bad file descriptor. Otherwise, it returns zero.

Because each program can have only a limited number of files open at any given time, programs that process many files should **close** files whenever possible.

Example

For an example of this function, see the entry for **open**.

See Also

creat, extended miscellaneous, open

Notes

When a program exits, **Let's C** automatically closes all files that had been opened via the STDIO function **fopen**. However, you must explicitly call **close** to close all files that had been opened with **open**, or the unclosed file will be truncated to zero bytes when the program exits.

LEXICON

cmp — Command

Compare bytes of two files
cmp [-ls] *file1 file2* **[***skip1 skip2***]**

cmp is a command that is included with **Let's C**. It compares *file1* and *file2* character by character, for equality. If *file1* is '**-**', **cmp** reads the standard input.

Normally, **cmp** notes the first difference and prints the line and character position, relative to any skips. If it encounters EOF on one file but not on the other, it prints the message "EOF on file*n*". The following are the options that can be used with **cmp**:

-l Each differing byte by printing the positions and octal values of the bytes of each file.

-s Print nothing, but return the exit status.

If the skip counts are present, **cmp** reads *skip1* bytes on *file1* and *skip2* bytes on *file2* before it begins to compare the two files.

The exit status is zero for identical files, one for non-identical files, and two for errors, e.g., bad command usage or inaccessible file.

See Also

commands

commands — Overview

Let's C includes a number of commands. They are listed below, with the command given on the left and a description on the right.

Commands included with **Let's C**:

cc	The compiler driver
cmp	Compare two files
cpp	The C preprocessor
egrep	String search utility
exetocom	Convert **.exe** files to **.com** files
fixobj	Edit object modules to allow cross-linking
make	Programming discipline
me	Microemacs screen editor
mwlib	Librarian for libraries in MS-DOS format
MWS	Mark Williams shell
nm	Print symbol tables
pr	Paginate text for printing
size	Print size of a file
strip	Remove debug tables from some executables
tail	Print the end of a file
wc	Count words/lines in ASCII files

Additional commands included with **Let's C Utilities**:

diff	Compare two files
ed	Line editor
m4	Macro processor
sort	Sort ASCII files
uniq	List/destroy duplicate lines

For more information on any of these commands, see its entry within the Lexicon.

See Also

DOS-specific features, MWS

comment — Definition

A *comment* is text that is embedded with a program but is ignored by the translator. It is intended to guide the reader of the code.

A comment is introduced by the characters **/***. The only exceptions are when these characters appear within a string literal or a character constant. In these instances, the characters **/*** have no special significance. When **/*** is read, all text is ignored until the characters ***/** are read. Once a comment is opened, the translator does nothing with the text except scan it for multi-byte characters and for the characters ***/** that close the comment.

The translator replaces a comment with a single white-space character; this is done during phase 3 of translation.

Cross-references

Standard, §3.1.9
The C Programming Language, ed. 2, p. 192

See Also

***/, /*, lexical elements, translation phases**

Notes

The Standard's definition of a comment does not allow comments to "nest." That is, you cannot have a comment within a comment. This may require that some code be revised. If you wish to exclude some code from translation temporarily, a sounder practice is to use the preprocessing directives **#ifdef** and **#endif**. For example,

```
#ifdef DEBUG
   . . .
#endif
```

will include code only if **DEBUG** has been defined as being a macro.

It is possible to open a comment inadvertently. For example, the code

```
int *intptr, int1, int2;
   . . .
int2 = int1/*intptr;
```

inadvertently creates a comment symbol out of the division operator '/' and the pointer-dereference operator '*'. *Caveat utilitor.*

compatible types — Definition

To judge whether two types are compatible, several factors must be considered.

Scalar types

First, the base types must be identical. Second, all specifiers must match, except for signedness (i.e., it does not matter whether either or both are signed or unsigned). Third, all type qualifiers must match. There are special semantics to determine whether qualified objects are compatible to ensure that qualified types are not "hidden". See the entry **type qualifiers** for more information.

Structures

For two structures to be compatible, they must have the same "tagged type". For example, the structures

```
FILE struct1;
FILE struct2;
```

are compatible, because the tagged type of each is **FILE**. On the other hand, in the following code

```
struct s1 { int s1_i } s1;
struct s2 { int s2_i } s2;
```

the structures **s1** and **s2** are not compatible.

Pointers

For two pointers to be compatible, they must point to the same type of object. Other pointers may be compatible if they are suitably cast.

Cross-reference

Standard, §3.1.2.6, §3.5.2-4

See Also

type specifier, types

compile —

To *compile* a program means to translate it with a compiler. A compiler is a translator that takes a set of high-level source instructions (i.e., C code) and produces a set of machine instructions that implement the behavior that the source instructions describe.

See Also

Definitions, interpret, link

compliance — Definition

Compliance refers to the degree to which a program and an implementation conform to the Standard's descriptions of the C language.

A *strictly conforming program* is one that uses only the features of the language and the library routines that are described within the Standard. It does not produce any behavior that is implementation defined, unspecified, or undefined. It does not exceed any minimum maximum set by the Standard. A strictly conforming program should be maximally portable to any environment for which a conforming implementation exists.

A *conforming program* is any program that can be translated by a conforming implementation. It may use library functions other than those described in the Standard, it may evoke non-Standard behavior, and it may use extensions to the language that are recognized by the implementation.

There are two varieties of *conforming implementation*: *conforming hosted implementation* and *conforming freestanding implementation*. A conforming hosted implementation is one that can translate any strictly conforming program. A conforming freestanding implementation is one that can translate any strictly conforming program whose use of macros and functions is restricted to those defined in the headers **float.h**, **limits.h**, **stdarg.h**, and **stddef.h**.

Cross-reference

Standard, §1.7

See Also

behavior, Definitions, limits

con — Operating system device

Logical device for the console

MS-DOS gives names to its logical devices. **Let's C** uses these names to allow its STDIO library routines to access these devices via MS-DOS. **con** is the logical device for the console.

Example

The following example demonstrates how to open the console device.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      FILE *fp, *fopen();
      if ((fp = fopen("con", "w")) != NULL)
            fprintf(fp, "con enabled.\n");
      else printf("con: cannot open.\n");
      return EXIT_SUCCESS;
}
```

See Also

aux, com1, lpt1, nul, operating system devices

const — C keyword

Qualify an identifier as not modifiable

The type qualifier **const** marks an object as being unmodifiable. An object declared as being **const** cannot be used on the left side of an assignment, or have its value modified in any way. Because of these restrictions, an implementation may place objects declared to be **const** into a read-only region of storage.

Judicious use of **const** allows the translator to optimize more thoroughly, for it does not have to include code to check whether the object has been modified.

Most of the prototypes for library functions use **const** to mark identifiers that are not modified by the function.

Cross-references

Standard, §3.5.3
The C Programming Language, ed. 2, p. 40

See Also

type qualifier, volatile

constant expressions — Definition

A *constant expression* is one that represents a constant. Constant expressions are required in a variety of situations: when the value of an enumeration constant is set; when the size of an array is declared; as a constant to be used in a **case** statement; or as the size of a bit-field declaration.

Every constant expression must return a value that is within the range representable by its type. No constant expression can contain assignment operators, increment or decrement operators, function calls, or the comma operator. The only exception is when it used as the operand to the operator **sizeof**.

LEXICON

The Standard describes the following varieties of constant expressions:

Address constant expression
This type of constant is an expression that points to an object or a function. The operators **[]**, *****, **&**, **.**, and **->** may be used to create an address constant, as may a pointer cast.

Arithmetic constant expression
This type of constant has an arithmetic type, and is one the following:

- character constant
- enumeration constant
- floating constant
- integer constant
- **sizeof** expression

  An arithmetic constant expression can be cast only to another arithmetic type, except when it is an operand to **sizeof**.

Integral constant expression
This type of constant has integral type, and is one of the following:

- character constant
- enumeration constant
- a floating constant that is the immediate operand of a cast.
- integer constant
- **sizeof** constant

When a constant expression is used to initialize a static variable, it must resolve, when translated, into one of the following:

- An address constant.
- An address constant for an object type, plus or minus an integral constant expression.
- An arithmetic constant expression.

Initializers on local variables that are not declared **static** are not so restrictive.

Cross-references

Standard, §3.4
The C Programming Language, ed. 2, p. 38

See Also

constants, expressions, initializers, Language, void expression

Notes

Constant expressions can be combined when translated. The precision and accuracy of such translation-time evaluation must be at least those of the execution environment. This requirement was designed with cross-compilers in mind, where the execution environment might differ from translation environment.

A constant expression may be resolved into a constant by the translator. Therefore, it can be used in any circumstance that calls for a constant. For this reason, the Standard forbids the use in an

#if statement in a constant expression that queries the run-time environment. A program that does include a **#if** statement that queries the environment will not run the same when translated by an ANSI-compatible translator.

constants — Overview

A *constant* is a lexical element that represents a set numerical value. The four categories of constants are as follows:

character constants	A character constant or wide-character constant
enumeration constants	A constant used in an **enum**
floating constants	A floating-point number
integer constants	An integer

Each type is determined by the form of the token. For example,

```
5L
```

defines a constant of type **long**, and

```
5.03
```

is a floating-point constant.

Cross-references

Standard, §3.1.3
The C Programming Language, ed. 2, pp. 192*ff*

See Also

constant expressions, lexical elements

continue — C keyword

Force next iteration of a loop
continue;

continue forces the next iteration of a **for**, **while**, or **do** loop. It works only upon the smallest enclosing loop.

continue forces a loop to iterate by jumping to the end of the loop, which is where iteration evaluation is made. For example, the code

```
while(statement) {
        . . .
      if (statement)
            continue;
        . . .
}
```

is equivalent to:

```
while(statement) {
        . . .
      if (statement)
            goto end;
        . . .
      end: ;
}
```

Example

For an example of this statement, see **mktime**.

LEXICON

Cross-references

Standard, §3.6.6.2
The C Programming Language, ed. 2, p. 64

See Also

break, C keywords, goto, return, statements

conversions — Definition

The term *conversion* means to change the type of an object, function, or constant, either explicitly or implicitly. Explicit conversion occurs when an object or function is cast to another type by a cast operator. Implicit conversion occurs when the type of the object or function is changed by an operator without a cast operator being used.

When an object or function is converted into a compatible type, its value does not change.

The following paragraphs summarize conversion for different types of objects.

Enumeration constants
These constants are always converted implicitly to **int**s.

Floating types
When a floating type is converted to an integral type, the fractional portion is thrown away. If the value of the integral part cannot be represented by the new type, behavior is undefined.

When a **float** is promoted to **double** or **long double**, its value is unchanged. Likewise, when a **double** is promoted to a **long double**, its value is unchanged.

A floating type may be converted to a smaller floating type. If its value cannot be represented by the new type, behavior is undefined. If its value lies within the range of values that can be represented by the smaller type but cannot be represented precisely, then its value is rounded to the next highest or next lowest value, depending upon the implementation.

Integral types
A **char**, a **short int**, an enumerated type, or a bit-field, whether signed or unsigned, may be used in any situation that calls for an **int**. The type to be promoted is converted to an **int** if an **int** can hold all of its possible values. If an **int** cannot hold all of its possible values, then it is converted to an **unsigned int**. This rule is called *integral promotion*. This conversion retains the value of the type to be promoted, including its sign. Thus, it is called a *value-preserving* promotion.

Some current implementations of C use a scheme for promotion that is called *unsigned preserving*. Under this scheme, an **unsigned char** or **unsigned short** is always promoted to **unsigned int**. Under certain circumstances, a program that depends upon unsigned-preserving promotion will behave differently when subjected to value-preserving promotion, and probably without warning. This is a quiet change that may break some existing code.

An integral type may be converted to a floating type. If its value lies within the range of values that can be represented by the floating type, but it cannot be represented precisely, then its value is rounded to the next highest or next lowest value, depending upon the implementation.

Signed and unsigned integers
The following rules apply when a signed or an unsigned integer is converted to another integral type:

- When a positive, signed integer is promoted to an unsigned integer of the same or larger type, its value is unchanged.
- When a negative integer is promoted to an unsigned integer of the same or larger type, it is first promoted to the signed equivalent of the unsigned type. It is then converted to unsigned by incrementing its value by one plus the maximum value that can be held by the unsigned type. On two's complement machines, the bit pattern of the promoted object does not changed. The only exception is that the sign bit is copied to fill any extra bits of new type, should it be larger than the old type.
- When a signed or unsigned integer is demoted to a smaller, unsigned type, its value is the non-negative remainder that occurs when the value of the original type is divided by one plus the maximum value that can be held by the smaller type.
- When a signed or unsigned integer is demoted to a smaller, signed type, if its value cannot be represented by the new type, the result is implementation-defined.
- When an unsigned integer is converted to a signed type of the same size, if its value cannot be represented by the new type, the result is implementation-defined.

Usual arithmetic conversions

Many binary operators convert their operands and yield a result of a type common to both. The rules that govern such conversions are called the *usual arithmetic conversions*. The following lists the usual arithmetic conversions. If two conflict, the rule *higher* in the list applies:

- If either operand has type **long double**, the other operand is converted to **long double**.
- If either operand has type **double**, the other operand is converted to **double**.
- If either operand has type **float**, the other operand is converted to **float**.
- If either operand has type **unsigned long int**, then the other operand is converted to **unsigned long int**.
- If one operand has the type **long int** and the other operand has type **unsigned int**, the other operand is converted to **long int** if that type can hold all of the values of an **unsigned int**. Otherwise, both operands are promoted to **unsigned long int**.
- If either operand has type **long int**, the other operand is converted to **long int**.
- If either operand has type **unsigned int**, the other operand is converted to **unsigned int**.
- If none of the above rules apply, then both operands have type **int**.

Cross-references

Standard, §3.2
The C Programming Language, ed. 2, pp. 197*ff*

See Also

explicit conversion, function designator, implicit conversion, integral promotions, Language, lvalue, null pointer constant, value preserving, void expression

Notes

The "as if" rule gives implementors some leeway in applying the rules for usual arithmetic conversions. For example, the conversion rules specify that operands of type **char** must first be widened to type **int** before the operation is performed; however, if the same result would be produced by performing the operation on **char** operands, then the operands need not be widened.

LEXICON

Because the Standard now allows single-precision floating-point arithmetic on **float** operands, some round-off error could occur. Casts will force the operands in question to be promoted, and the operation to be carried out with the wider type.

cos() — Mathematics (libm)

Calculate cosine
#include <math.h>
double cos(double *radian***);**

cos calculates and returns the cosine of its argument *radian*, which must be expressed in radians.

Example

For an example of this function, see **sin**.

Cross-references

Standard, §4.5.2.5
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan, atan2, mathematics, sin, tan

cosh() — Mathematics (libm)

Calculate hyperbolic cosine
#include <math.h>
double cosh(double *value***);**

cosh calculates and returns the hyperbolic cosine of *value*. A range error will occur if the argument is too large.

Cross-references

Standard, §4.5.3.1
The C Programming Language, ed. 2, p. 251

See Also

mathematics, sinh, tanh

cpp — Command

C preprocessor
cpp [*option…***] [***file…***]**

The command **cpp** calls the preprocessor/parser **cc0** to perform C preprocessing on C programs. It performs the operations described in section 3.8 of the Standard; these include file inclusion, conditional code selection, constant definition, and macro definition. See the entry on **preprocessing** for a full description of the C's preprocessing language.

cpp reads each input *file*, or **stdin** if no file is specified; processes directives, and writes its product on **stdout**. If the option **-E** is not used, **cpp** also writes into its output statements of the form **#***n filename*, so that the parser will be able to connect its error messages and debugger output with the original line numbers in your source files.

Options

The following summarizes **cpp**'s options:

-D*VARIABLE*
Define *VARIABLE* for the preprocessor at compilation time. For example, the command

```
cc -DLIMIT=20 foo.c
```

tells the preprocessor to define the variable **LIMIT** to be 20. The compiled program acts as though the directive **#define LIMIT 20** were included before its first line.

-E Strip all comments and line numbers from the source code. This option is used to preprocess assembly-language files or other sources, and should not be used with the other compiler phases.

-I *directory*
C allows two types of **#include** directives in a C program, i.e., **#include "file.h"** and **#include <file.h>**. The **-I** option tells **cpp** to search a specific directory for the files you have named in your **#include** directives, in addition to the directories that it searches by default. You can have more than one **-I** option on your **cc** command line.

-o *file*
Write output into *file*. If this option is missing, **cpp** writes its output onto **stdout**, which may be redirected.

-U*VARIABLE*
Undefine *VARIABLE*, as if an **#undef** directive were included in the source program. This is used to undefine the variables that **cpp** defines by default.

See Also

cc, preprocessing

creat() — Extended function (libc)

Create/truncate a file
short creat(char **file***, short** *mode***);**

creat creates a new *file* or truncates an existing *file*. It returns a file descriptor that identifies *file* for subsequent system calls. If *file* already exists, its contents are erased.

creat ignores its *mode* argument. This argument exists for compatibility with implementations of **creat** under UNIX and related operating systems.

If the call is successful, **creat** returns a file descriptor. It returns **-1** if it could not create the file, typically because of insufficient system resources, or nonexistent path.

Example

For an example of this routine, see the entry for **open**.

See Also

extended miscellaneous, fopen, fdopen

csreg() — i8086 support (libc)

Get value from CS register
#include <dos.h>
unsigned csreg(void)

csreg returns the value from the i8086 CS register, which points to the base of the code segment.

Example

The following example uses the functions **csreg**, **dsreg**, **esreg**, and **ssreg** to print the contents of the segment registers.

LEXICON

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      printf("csreg=%04x\n", csreg());
      printf("dsreg=%04x\n", dsreg());
      printf("esreg=%04x\n", esreg());
      printf("ssreg=%04x\n", ssreg());
      return EXIT_SUCCESS;
}
```

See Also

dsreg, esreg, i8086 support, ssreg

ctime() — Time function (libc)

Convert calendar time to text
#include <time.h>
char *ctime(const time_t ****timeptr*);**

The function **ctime** reads the calendar time pointed to by *timeptr*, and converts it into a string of the form

```
Tue Dec 10 14:14:55 1987\n\0
```

ctime is equivalent to:

```
asctime(localtime(timeptr));
```

timeptr points to type **time_t**, which is defined in the header **time.h**.

Example

This example displays the current time.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void)
{
      time_t t;
      time(&t);

      printf(ctime(&t));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.3.2
The C Programming Language, ed. 2, p. 256

See Also

asctime, date and time, gmtime, localtime, strftime, time_t

ctype.h — Header

Header for character-handling functions
#include <ctype.h>

ctype.h is the header file that declares the functions used to handle characters. These are as follows:

isalnum	Check if a character is a numeral or letter
isalpha	Check if a character is a letter
iscntrl	Check if a character is a control character
isdigit	Check if a character is a numeral
isgraph	Check if a character is printable
islower	Check if a character is a lower-case letter
isprint	Check if a character is printable
ispunct	Check if a character is a punctuation mark
isspace	Check if a character is white space
isupper	Check if a character is a upper-case letter
isxdigit	Check if a character is a hexadecimal numeral
tolower	Convert character to lower case
toupper	Convert character to upper case

Cross-references

Standard, §4.3
The C Programming Language, ed. 2, p. 248

See Also

character handling, header, xctype.h

D

daemon — Definition

A **daemon**, in the context of C programming, is a process that is designed to perform a particular task or control a particular device without requiring the intervention of a human operator.

See Also

Definitions, process

date and time — Overview

#include <time.h>

The Standard describes nine functions that can be used to represent date and time, as follows:

Time conversion

asctime	Convert broken-down time to text
ctime	Convert calendar time to text
gmtime	Convert calendar time to Universal Coordinated Time
localtime	Convert calendar time to local time
strftime	Format locale-specific time

Time manipulation

clock	Get processor time used by the program
difftime	Calculate difference between two times
mktime	Convert broken-down time into calendar time
time	Get current calendar time

These functions use the following structures:

clock_t	System time
time_t	Calendar time
tm	Broken-down time

Let's C defines **time_t** as a 32-bit number that holds the number of seconds since January 1, 1970, 0h00m00s UTC.

The structure **tm** is defined as follows:

```
typedef struct tm {
      int tm_sec; /* second [0-60] */
      int tm_min; /* minute [0-59] */
      int tm_hour;/* hour [0-23]: 0 = midnight */
      int tm_mday;/* day of the month [1-31] */
      int tm_mon; /* month [0-11]: 0=January */
      int tm_year;/* year since 1900 A.D. */
      int tm_wday;/* day of week [0-6]: 0=Sunday */
      int tm_yday;/* day of the year [0-366] */
      int tm_isdst;/* daylight savings time flag */
} tm_t;
```

The member **tm_sec** can hold 61 seconds. This is done so that it can hold the "leap seconds" that are used internationally to help coordinate atomic clocks with pulsars and solar time.

Finally, the manifest constant **CLK_TCK** is used to convert the value returned by the function **clock** into seconds of real time. It is defined as being equivalent to one tick of the system clock. On the IBM PC and compatibles, this is equivalent to 18.206481933 milliseconds. This value does *not* change on machines that run at speeds higher than the standard 4.77 megahertz.

To print the local time, a program must perform the following tasks: First, read the system time with

time. Then, it must pass **time**'s output to **localtime**, which breaks it down into the **tm** structure. Next, it must pass **localtime**'s output to **asctime**, which transforms the **tm** structure into an ASCII string. Finally, it must pass the output of **asctime** to **printf**, which displays it on the standard output.

Let's C also includes numerous extensions to the ANSI Standard's time functions. These extensions increase the scope and accuracy of the Standard, and ease calculation of some time elements. See the entry on **extended time** for more information.

Cross-references

Standard, §4.12
The C Programming Language, ed. 2, pp. 255*ff*

See Also

broken-down time, calendar time, daylight saving time, extended time, local time, Library, TIMEZONE, universal coordinated time

dayspermonth() — Extended function (libc)

Return number of days in a given month
#include <xtime.h>
int dayspermonth(int *month***, int** *year***);**

dayspermonth returns the number of days in a given month of a given year A.D. *month* is the number of the month in question, from one to 12. *year* is the year A.D. in which *month* appears. Note that there is no year 0.

See Also

extended time, isleapyear, xtime.h

Notes

To conform to the ANSI Standard, this function has been moved from the header **time.h** to the header **xtime.h**. This may require that some code be rewritten.

DBL_DIG — Manifest constant

#include <float.h>
DBL_DIG is a manifest constant that is defined in the header **float.h**. It is an expression that defines the number of decimal digits of precision representable in an object of type **double**. It is defined to be ten.

Cross-references

Standard, §2.2.4.2
The C Programming Language, ed. 2, p. 258

See Also

float.h, numerical limits

decimal-point character — Definition

The *decimal-point character* as being the character that marks the beginning of the fraction in a floating-point number. How this character is represented depends upon the program's locale. The locale specifier **LC_NUMERIC** describes how a particular locale represents the decimal-point character. In the **C** locale, it is the period '.'.

This character is used by the functions that convert a floating-point number to a string, or read a string and convert it to a floating-point number, i.e., **atof**, **fprintf**, **fscanf**, **printf**, **scanf**, **sprintf**, **sscanf**, **strtod**, **vfprintf**, **vprintf**, and **vsprintf**. This character is *not* used within C source; for

example,

```
sqrt(1,2);
```

passes two integer constants to **sqrt**, even if ',' is the decimal-point character for the current locale. Therefore, to print a C source file use the **C** locale, even if the program establishes another locale.

Cross-reference

Standard, §4.1.1

See Also

Definitions, localization

declarations — Overview

A *declaration* gives the type, storage class, linkage, and scope of a given identifier.

If a declaration also causes storage to be allocated for the object declared, then it is called a *definition*.

Declarators may be within a list, separated by commas. Each declarator has the type given at the beginning of the list, although a declarator may also have additional type information. For example,

```
int example1, *example2;
```

declares two variables: **example1** has type **int**, whereas **example2** has type "pointer to **int**."

Objects may be initialized when they are declared. See **initialization** for more information.

Cross-references

Standard, §3.5
The C Programming Language, ed. 2, pp. 210*ff*

See Also

bit-fields, declarators, definition, initialization, Language, linkage, scope, storage-class specifiers, type qualifiers, type specifiers

declarators — Overview

A *declarator* consists of an object being declared plus its array, pointer, and function modifiers.

For example,

```
int arrayname[10];
```

declares an array.

```
int functionname( int arg1, int arg2, char *arg3 );
```

declares a function.

```
int *pointername;
```

declares a pointer.

An implementation must be able to support at least 12 levels of declarators. Most implementations had given a lower limit.

Cross-references

Standard, §3.5.4
The C Programming Language, ed. 2, pp. 215*ff*

See Also

array declarators, declarations, function declarators, pointer declarators

Notes

To clarify some terminology that may be confusing:

A *declaration* encompasses the object declared, plus its specifiers, qualifiers, and levels of declarators.

A *declarator* consists of the object declared, plus its levels of specifiers (which set array dimensions, functions, or pointers).

A *definition* is a declaration that allocates storage.

default — C keyword

Default entry in switch table

default is a label that marks the default entry in the body of a **switch** statement. If none of the **case** labels match the value of the **switch** statement's conditional expression, then the **switch** statement jumps to the point marked by the **default** label, and begins execution from there.

Example

For an example of this label, see **printf**.

Cross-references

Standard, §3.6.1
The C Programming Language, ed. 2, p. 58

See Also

C keywords, case, statements, switch

Notes

A **switch** statement is not required to include a **default** label, but it is good programming practice to include one.

defined — C keyword

Check if identifier is defined
defined(*identifier* **)**
defined *identifier*

The Standard describes a new C keyword, **defined**. This keyword is used to check if *identifier* has been defined as macro or manifest constant. The preprocessing directives

```
#if defined(identifier)
```

and

```
#if defined identifier
```

have exactly the same effect as the directive:

```
#ifdef identifier
```

The **defined** operator is permitted only within **#if** and **#elif** expressions. It may not be used in any other context.

defined is not a reserved word. It can be used in more complex conditional statements, i.e.:

```
#if LEVEL==3 && defined FOO
```

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#if, #ifdef, keywords, preprocessing

definition — Definition

A *definition* is a declaration that also allocates storage for the item declared. For example,

```
int example[];
```

declares that **example** names an array of **int**s. Because the declaration does not say how large of an array **example** is, no memory is reserved; thus, this is a declaration but not a definition.

However, the declaration

```
int example[10];
```

declares that **example** names an array of ten **int**s. Because the declaration states how large **example** is, an appropriately sized portion of memory is reserved for it. Thus, this declaration is also a definition.

declaration and *definition* are easily confused, because the words are used in ways that are somewhat contrary to their normal English meanings.

A function definition is a special kind of definition that operates by its own rules. See **function definition** for more details.

Cross-references

Standard, §3.5
The C Programming Language, ed. 2, pp. 201, 210

See Also

declarations, function definition

Definitions — Overview

These definitions apply to topics throughout this Lexicon:

address
alias
alignment
argument
arena
ASCII
behavior
BIOS
bit
bit-fields
bit map
block
buffer
byte
compliance
cc0
cc1

cc2
cc3
cc4
daemon
decimal-point character
directory
domain error
executable file
false
field
file
file descriptor
interrupt
letter
link
manifest constant
nested comments
nybble
object format
object
parameter
pattern
port
portability
process
pun
quiet change
random access
range error
ranlib
read-only memory
record
register
rvalue
spirit of C
stack
Standard
standard error
standard input
standard output
stream
string
true
Universal Coordinated Time
wildcards

Cross-references

Standard, §1.6

See Also

LEXICON

diagnostics — Overview

The term *diagnostics* has two meanings in the ANSI Standard. The first is a set of macros that are used to test an expression at run time. The second refers to the way **Let's C** warns a user that a program contains an error.

Run-Time Diagnostics

The Standard describes a mechanism whereby an expression can be tested at run time. The macro **assert** tests the value of a given expression as the program runs. If the expression is false, **assert** prints a message into the standard error stream and then calls **abort**.

assert is defined in the header **assert.h**. This header also defines the manifest constant **NDEBUG**. If you define this macro before including **assert.h**, **assert** is redefined as follows:

```
#define assert(ignore)
```

This turns off **assert**. If an expression evaluated by **assert** has any side effects, using **NDEBUG** will change the program's behavior.

Diagnostic Warnings

Let's C produces a diagnostic for every translation unit that contains one or more errors of syntax rules or syntax constraints. A diagnostic can be either a fatal error, which prints a message and aborts translation, or simply a warning that prints a message and allows translation to proceed.

Cross-reference

Standard, §2.1.1.3

See Also

Library

difftime() — Time function (libc)

Calculate difference between two times
#include <time.h>
double difftime(time_t *newtime*, **time_t** *oldtime*);

difftime subtracts *oldtime* from *newtime*, and returns the difference in seconds.

Both arguments are of type **time_t**, which is defined in the header **time.h**.

Example

This example uses **difftime** to show an arbitrary time difference.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void)
{
      time_t      t1, t2;

      time(&t1);
      printf("Press enter when you feel like it.\n");
      getchar();
      time(&t2);

      printf("You waited %f seconds\n", difftime(t2, t1));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.2.2
The C Programming Language, ed. 2, p. 256

See Also

clock, date and time, mktime, time_t

digit — Definition

A *digit* is any of the following characters:

```
0  1  2  3  4  5  6  7  8  9
```

Cross-reference

Standard, §3.1.2

See Also

identifiers, nondigit

directory — Definition

A **directory** is a table that maps names to files. In other words, it associates the names of a file with their locations on the mass storage device. Under some operating systems, directories are also files, and can be handled like a file.

Directories allow files to be organized on a mass storage device in a rational manner, by function or owner.

See Also

Definitions, file, path

div() — General utility (libc)

Perform integer division
#include <stdlib.h>
div_t div(int *numerator*, **int** *denominator*);

div divides *numerator* by *denominator*. It returns a structure of the type **div_t**. This structure consists of two **int** members, one named **quot** and the other **rem**. **div** writes the quotient into **quot** and the remainder into **rem**.

The sign of the quotient is positive if the signs of the arguments are the same; it is negative if the signs of the arguments differ. The sign of the remainder is the same as the sign of the numerator.

If the remainder is non-zero, the magnitude of the quotient is the largest integer less than the magnitude of the algebraic quotient. This is not guaranteed by the operators **/** and **%**, which merely do what the machine implements for divide.

Example

For an example of this function, see **memchr**.

Cross-references

Standard, §4.10.6.2
The C Programming Language, ed. 2, p. 253

See Also

/, div_t, general utilities, ldiv

LEXICON

Notes

The Standard includes this function to permit a useful feature found in most versions of FORTRAN, where the sign of the remainder will be the same as the sign of the numerator. Also, on most machines, division produces a remainder. This allows a quotient and remainder to be returned from one machine-divide operation.

If the result of division cannot be represented (e.g., because *denominator* is set to zero), the behavior of **div** is undefined.

div_t — Type

Type returned by **div()**
#include <stdlib.h>

div_t is a typedef that is declared in the header **stdlib.h**. It is the type returned by the function **div**.

div_t is a structure that consists of two **int** members, one named **quot** and the other **rem**. **div** writes its quotient into **quot** and its remainder into **rem**.

Example

For an example of using this type in a program, see **memchr**.

Cross-references

Standard, §4.10.6.2
The C Programming Language, ed. 2, p. 253

See Also

div, general utilities, integer arithmetic, stdlib.h

do — C keyword

Loop construct
do { *statement* } **while(***condition***);**

do establishes conditional loop. Unlike the loops established by **for** and **while**, the condition in a **do** loop is evaluated *after* the operation is performed. This guarantees that at least one iteration of the loop will be executed.

do always works in tandem with **while**. For example

```
do {
      puts("Next entry? ");
      fflush(stdout);
} while(getchar() != EOF);
```

prints a prompt on the screen and waits for the user to reply. The **do** loop is convenient in this instance because the prompt must appear at least once on the screen before the user replies.

Cross-references

Standard, §3.6.5.2
The C Programming Language, ed. 2, p. 63

See Also

break, C keywords, continue, for, statements, while

dos.h — Header

Define MS-DOS functions and devices
#include <dos.h>

dos.h is the header that defines MS-DOS functions and devices. It is used with functions that directly interface with MS-DOS, such as **intcall**.

See Also

header, intcall, signals/interrupts

DOS-specific features — Overview

Let's C includes many features that relate specifically to the IBM PC, including the following:

- Source code
- Commands to be used with **Let's C**
- Example programs

This manual also includes a number of articles that given information about the i8086 and MS-DOS. See the Lexicon entry **technical information** for a list of these articles.

See Also

Lexicon, archive, command, example, technical information

double — C keyword

A **double** is a data type that represents a double-precision floating-point number. It is defined as being at least as large as a **float** and no larger than a **long double**.

Like all floating-point numbers, a **double** consists of one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *mantissa*, or the number upon which the exponent works. The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the fraction will be increased. The format of a **double** and the range of values that it can encode are set in the following macros, all of which are defined in the header **limits.h**:

DBL_DIG
: This holds the number of decimal digits of precision. This must be at least ten.

DBL_EPSILON
: Where *b* indicates the base of the exponent (default, two) and *p* indicates the precision (or number of base *b* digits in the mantissa), this macro holds the minimum positive floating-point number *x* such that 1.0 + *x* does not equal 1.0, *b*^1-*p*. This must be at least 1E-9.

DBL_MAX
: This holds the maximum representable floating-point number. It must be at least 1E+37.

DBL_MAX_EXP
: This is the maximum integer such that the base raised to its power minus one is a representable floating-point number.

DBL_MAX_10_EXP
: This holds the maximum integer such that ten raised to its power is within the range of representable finite floating-point numbers. It must be at least +37.

DBL_MANT_DIG
: This gives the number of digits in the mantissa.

DBL_MIN
This gives the minimum value encodable within a **double**. This must be at least 1E-37.

DBL_MIN_EXP
This gives the minimum negative integer such that when the base is raised to that power minus one is a normalized floating-point number.

DBL_MIN_10_EXP
This gives the minimum negative integer such that ten raised to that power is within the range of normalized floating-point numbers. It must be at least -37.

For information on common floating-point formats, see **float**.

Cross-references

Standard, §2.2.4.2, §3.1.2.4, §3.1.3.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

float, long double, types

dsreg() — i8086 support (libc)

Get value from DS segment register
#include <dos.h>
unsigned dsreg(void)

dsreg returns the value from from the i8086 DS register, which points to the base of the data segment.

Example

For an example of this function, see the entry for **csreg**.

See Also

csreg, esreg, i8086 support, ssreg

dup() — Extended function (libc)

Duplicate a file descriptor
short dup(short *fd***);**

dup duplicates the existing file descriptor *fd*, and returns the new descriptor. The returned value is the smallest file descriptor that is not already in use by the calling process. It returns a negative number when an error occurs, such as a bad file descriptor or no file descriptor available.

See Also

dup2, extended miscellaneous, fdopen, open

dup2() — Extended function (libc)

Duplicate a file descriptor
short dup2(short *fd***,** *newfd***);**

dup2 duplicates the file descriptor *fd*. Unlike its cousin **dup**, **dup2** allows you to specify a new file descriptor *newfd*, rather than have the system select one. If *newfd* is already open, the system closes it before assigning it to the new file. **dup2** returns the duplicate descriptor. It returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

See Also

dup, extended miscellaneous, fdopen, open

E

ecvt() — Extended function (libc)

Convert floating-point numbers to strings
char *ecvt(double *d*, **int** *prec*, **int** **dp*, **int** ***signp);**

ecvt converts *d* into a null-terminated ASCII string of numerals with the precision of *prec*. Its operation resembles that of **printf**'s **%e** operator. **ecvt** rounds the last digit and returns a pointer to the result. On return, **ecvt** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string, to the right if positive, to the left if negative. It sets *signp* to point to an integer that indicates the sign of *d*, zero if positive and nonzero if negative.

Example

The following program demonstrates **ecvt**, **fcvt**, and **gcvt**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* prototypes for extended functions */
extern char *ecvt(double d, int prec, int *dp, int *signp);
extern char *fcvt(double d, int w, int *dp, int *signp);
extern char *gcvt(double d, int prec, char *buffer);

main(void)
{
      char buf[64];
      double d;
      int i, j;
      char *s;

      d = 1234.56789;
      s = ecvt(d, 5, &i, &j);
      /* prints ecvt="12346" i=4 j=0 */
      printf("ecvt=\"%s\" i=%d j=%d\n", s, i, j);

      strcpy(s, fcvt(d, 5, &i, &j));
      /* prints fcvt="123456789" i=4 j=0 */
      printf("fcvt=\"%s\" i=%d j=%d\n", s, i, j);

      s = gcvt(d, 5, buf);
      /* prints gcvt="1234.56789" */
      printf("gcvt=\"%s\"\n", s);

      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, fcvt, frexp, gcvt, ldexp, modf, printf

Notes

ecvt performs conversions within static string buffers that are overwritten by each execution.

egrep — Command

Extended pattern search
egrep [*option …***] [***pattern***] [***file …***]**

The command **egrep** searches each *file* for occurrences of *pattern* (also called a regular expression). If no *file* is specified, it searches what is typed into the standard input. Normally, it prints each line

matching the *pattern*.

Wildcards

The simplest *patterns* accepted by **egrep** are ordinary alphanumeric strings. **egrep** can also process *patterns* that include the following wildcard characters:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

***** Match zero or more repetitions of preceding character.

. Match any character except newline.

[*chars***]**
Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars***]**
Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

*****c Disregard special meaning of character *c*.

| Match the preceding pattern *or* the following pattern. For example, the pattern **cat | dog** matches either **cat** or **dog**. A newline within the *pattern* has the same meaning as '|'. Under MS-DOS, the '|' has special meaning, and must be enclosed within apostrophes.

+ Match one or more occurrences of the immediately preceding pattern element; it works like '*', except it matches at least one occurrence instead of zero or more occurrences.

? Match zero or one occurrence of the preceding element of the pattern.

(...) Parentheses may be used to group patterns. For example, **(Ivan)+** matches a sequence of one or more occurrences of the four letters 'I' 'v' 'a' or 'n'.

Because the metacharacters '*' and '?', are also special to MS-DOS, patterns that contain those literal characters must be quoted by enclosing *pattern* within double quotation marks.

Options

The following lists the available options:

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print how many lines match, rather than the lines themselves.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file that contains a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-l Print the name of each file that contains the string, rather than the lines themselves. This is useful when you are constructing a batch file.

-n When a line is printed, also print its number within the file.

-s Suppress all output, just return exit status.

LEXICON

-v Print a line only if the pattern is *not* found in the line.

-y Lower-case letters in the pattern match lower-case *and* upper-case letters on the input lines. A letter escaped with '\' in the pattern must be matched in exactly that case.

Diagnostics

egrep returns an exit status of zero for success, one for no matches, and two for error.

See Also

commands

Notes

egrep uses a deterministic finite automaton (DFA) for the search. It builds the DFA dynamically, so it begins doing useful work immediately. This means that **egrep** is considerably faster than earlier pattern-searching commands, on almost any length of file.

else — C keyword

Conditionally execute a statement
else *statement*;

else is the flip side of **if**: if the condition described in the **if** statement equals zero, then the statement introduced by **else** is executed. If, however, the condition described in the **if** statement is nonzero, then the statement it introduces is executed and the statement introduced by **else** is ignored.

An **else** statement is associated with the first preceding **else**-less **if** statement that is within the same block, but not within an enclosed block. For example,

```
if(conditional1) {
        if(conditional2)
                statement1
} else
        statement2
```

the **else** is associated with the **if** statement that uses *conditional1*, not the one that uses *conditional2*. On the other hand, in the code

```
if(conditional1)
        if(conditional2)
                statement1
else
        statement2
```

which does not use braces, the **else** is associated with the **if** statement that uses *conditional2*, not the one that uses *conditional1*.

Example

For an example of this statement, see **exit**.

Cross-references

Standard, §4.6.4.1
The C Programming Language, ed. 2, pp. 55*ff*

See Also

if, statements, switch

enum — C keyword

Enumerated data type
enum *identifier* **{** *enumerations* **}**

An **enum** is a data type whose possible values are limited to a set of constants.

For example,

```
enum opinion { yes, no, maybe };
```

declares type **opinion** to have one of three constant values; these are identified by the members **yes**, **no**, and **maybe**.

The translator assigns values to the identifiers from left to right, beginning with zero and increasing by one for each successive term. In the above example, the values of **yes**, **no**, and **maybe** are set, respectively, to zero, one, and two. Thus, the following example

```
enum opinion guess;
   . . .
guess = no;
```

sets the value of **guess** to one.

All enumerated identifiers must be distinct from all other identifiers in the program. The identifiers act as constants and are used wherever constants are appropriate.

If a member of an enumeration is followed by an equal sign and an integer, the identifier is assigned the given value and subsequent values increase by one from that value. For example,

```
enum opinion {yes, no=50, maybe};
```

sets the values of the members **yes**, **no**, and **maybe** to zero, 50, and 51, respectively. More than one enumerator can have the same value. For example:

```
enum opinion {yes, no=50, nah=50, nope=50, maybe};
```

assigns duplicate values to the members **no**, **nah**, and **nope**.

An enumeration constant always has type **int**.

Cross-references

Standard, §3.5.2.2
The C Programming Language, ed. 2, p. 39

See Also

type specifier

Notes

Prior to the introduction of enumerated data types in C, programmers would create lists of manifest constants whose values took the values that enumerated constants now take.

Unlike more strongly typed languages, in which enumerated constants are checked to ensure that they are part of the specified set of values, **enum**s in C are only required to be of type **int**. No additional checking is performed on enumeration constants.

enumeration constant — Definition

An *enumeration constant* is a member of an enumeration. This constant has type **int**.

For example, in the enumeration

```
enum example { blue, green, yellow };
```

blue is an enumeration constant.

Cross-references

Standard, §3.1.3.3
The C Programming Language, ed. 2, pp. 39, 194

See Also

constants, enum

environmental variable — Overview

An *environmental variable* is a variable that is set through the operating system, and which a program can read at run time. These variables are most commonly used to change the way a program behaves.

Let's C uses the following environmental variables in its operation:

CCHEAD	Variables at head of compilation command
CCTAIL	Variables at tail of compilation command
INCDIR	Directory that holds include files
LIBPATH	Directories that hold libraries
PATH	Directories that hold executable files
TIMEZONE	Time zone information
TMPDIR	Directory that holds temporary files

Because of the limited environment space available under many version of MS-DOS, the variables **INCDIR**, **LIBPATH**, and **TMPDIR** often are not set. Instead, their information is placed into the file **ccargs**, which is built automatically when you install **Let's C**. You need to set the variable **TIMEZONE** only if you are writing programs that need exact time zone information.

See Also

Environment

envp — Definition

Argument passed to main
char *envp[];

envp is an abbreviation for **env**ironmental **p**arameter. It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function, and is by convention the third argument passed to **main**.

The MS-DOS runtime startup routines always set **envp** to NULL, i.e., no **envp** is passed. **Let's C** calls **main(***argc***,** *argv***, NULL)**; however, **envp** is significant under some other operating systems, including TOS, UNIX, and COHERENT.

See Also

argc, argv, Environment, main

EOF — Manifest constant

Indicate end of a file
#include <stdio.h>

EOF is an indicator that is returned by several STDIO functions to indicate that the current file position is the end of the file. Its value is defined by the implementation, but a common value is -1 on many systems, which is also a common error return.

The actual bytes used to delineate the end of a file may vary between implementations.

Many STDIO functions, when they read **EOF**, set the end-of-file indicator that is associated with the stream being read. Before more data can be read from the stream, its end-of-file indicator must be cleared. Resetting the file-position indicator with the functions **fseek**, **fsetpos**, or **ftell** will clear the indicator, as will returning a character to the stream with the function **ungetc**.

Example

For an example of this macro in a program, see **tmpfile**.

Cross-references

Standard, §4.3, §4.9.1; Rationale, §4.3
The C Programming Language, ed. 2, p. 151

See Also

file, stream, STDIO, stdio.h

errno — Macro

External integer that holds error status
#include <errno.h>

errno is a macro that is defined in the header **errno.h**. It expands to a global integer of type **volatile int**.

When a program begins to execute, **errno** is initialized to zero. Thereafter, whenever a mathematics function or other library function wishes to return information about any error that occurs during its operation, it writes the appropriate error number into **errno**, where it can be read either by the environment or by another function.

The functions **perror** and **strerror** can be used to translate the contents of **errno** into a text message.

Example

For an example of using this macro in a program, see **vfprintf**.

Cross-references

Standard, §4.1.3
The C Programming Language, ed. 2, p. 248

See Also

EDOM, ERANGE, errno.h, errors, mathematics

Notes

Only certain library functions set **errno**, and then only if certain error conditions occur. Remember that it is your responsibility to clear **errno** before the function in question is called. Other functions may also set **errno**.

LEXICON

Although it is widely believed that a program that checks the value of **errno** after each function is more portable than one that does not, this is not necessarily true. Some implementations use in-line expansion of library function to speed execution, and so forego the use of **errno**. The cautious programmer is best advised to check the value of input arguments before calling a library function and, of course, to check its return value before checking **errno**.

errno.h — Header

Define errno and error codes
#include <errno.h>

errno.h is a header that holds information which relates to the reporting of error conditions. It defines the macro **errno**, which expands to global variable of type **volatile int**. If an error condition occurs, a function can write a value into **errno**, to report just what type of error occurred.

For a list of the MS-DOS system errors described in **errno.h**, see **error**codes**.**

Cross-references

Standard, §4.1.3
The C Programming Language, ed. 2, p. 248

See Also

errno, error codes, errors

escape sequences — Definition

An *escape sequence* is a set of characters that, together, represent one character that may have a special significance. The Standard recognizes the following escape sequences:

\'	Literal apostrophe
\"	Literal quotation mark
\?	Literal question mark
****	Literal backslash
\a	Alert; ring the bell or print visual alert
\b	Horizontal backspace
\f	Form feed; force output device to begin a new page
\n	Newline; move to next line
\r	Carriage return; move to beginning of line
\t	Horizontal tabulation; move to next tabulation mark
\v	Vertical tabulation; move to next tabulation mark
*****NNN*	Octal number
\x*NN*	Hexadecimal number

An escape sequence may be embedded within a character constant or a string literal. In a string literal, the apostrophe may be represented either by itself or by its escape sequence, whereas in a character constant the quotation mark may be represented by itself or by its escape sequence.

Two question marks together may introduce a trigraph, which is interpreted even within a string literal. If you want to print two literal question marks, use the escape sequence **\?\?**. For more information, see **trigraph sequences**.

The escape sequences **\a** through **\v** let you use characters that control the output device.

A backslash followed by one, two, or three octal digits encodes an octal number. For example, in ASCII implementations of C, the escape sequence **'\141'** encodes the octal value 141 into an **int**-length object. When interpreted under an environment that uses ASCII, this prints the letter 'a'. Likewise, the escape sequence **\x** followed by an arbitrary number of hexadecimal digits encodes a

hexadecimal number.

Example

The following example demonstrates the use of the escape sequence **\b**, which prints a backspace character. It prints a message, backspaces over it, and then prints another message.

```
#include <stdio.h>
main()
{
      printf("BLINK!\b\b\b\b\b\bhello, world\n");
}
```

Cross-references

Standard, §2.2.2, §3.1.3.4
The C Programming Language, ed. 2, p. 193

See Also

character constant, constants, string literal, trigraph sequences

Notes

Previous releases of **Let's C** defined the escape sequences **\a** and **\x** differently.

Some implementations of C permit the digit '8' to be used with an octal number. For example, the character constant **'\078'** is regarded by these implementations as being equivalent to octal 100. Under ANSI C, **'\078'** will be interpreted as representing octal 7 plus the character constant **'8'**. This, too, is a quiet change that may break some existing code.

The escape sequence '\0' is used by many existing implementations to represent the null character.

esreg() — i8086 support (libc)

Get value from ES segment register
#include <dos.h>
unsigned esreg(void)

esreg returns the value from from the i8086 ES register, which points to the base of the "extra" segment. In SMALL model, this register always holds the same value as the DS register.

Example

For an example of this function, see the entry for **csreg**.

See Also

csreg, dsreg, i8086 support, ssreg " ENVIRONMENTS: LC

exargs() — Extended miscellaneous (libc)

Get and parse a command line
int exargs(char **name*, **int** *argc*, **char** **argv*[], **char** **xargv*[], **int** *maxarg*);

exargs provides a uniform mechanism by which programs that are run under MS-DOS can read and parse command lines. It cooperates with the C runtime startup to be as transparent as possible to the user.

The parameters *argc* and *argv* are the usual parameters to **main**. They are parsed from the MS-DOS command tail by **_main** in the C runtime startup routine. **exargs** simply takes all of a command's arguments from *argv*.

exargs parses command lines by breaking them into a list of arguments separated by white space (i.e., a space or tab character). It expands wildcard arguments, writes pointers to the arguments

into the array *xargv*, and returns the number of arguments. **exargs** then puts a **NULL** pointer at the end of the list, so *xargv* looks much like the *argv* parameter to **main**. *maxarg* is the maximum number of arguments that a command can take, that is, the maximum number that will fit into the array *xargv*.

exargs interprets a command line of the form **@***name* as a file reference: it opens the file *name* and reads command lines from it. Such files can also contain references to yet other files.

exargs uses **getenv** to search the environment for the strings *name***HEAD** and *name***TAIL**. If found, it adds the value of *name***HEAD** at the beginning of the argument list and the value of *name***TAIL** at the end. For example, the **cc** command uses **exargs** with a *name* argument of **cc**; accordingly, it looks for **CCHEAD** and **CCTAIL** in the environment to provide command-specific information.

exargs returns the size of its argument list, which is suitable for assignment to **argc**.

Example

The following function converts UNIX and COHERENT utilities to MS-DOS utilities by changing **argc** and **argv** via **exargs**.

```
#define MAXARGS 1023
#include <stdio.h>
#include <stdlib.h>

void
msdoscvt(int *argc, char *name, ***argv)
{
      char **xargv;

      if(NULL == (xargv = malloc((MAXARGS + 1)
            * sizeof(char *))))
            abort();
      *argc = exargs(name, *argc, *argv, &xargv[1], MAXARGS) + 1;
      xargv[0] = name;
      *argv = realloc(xargv, (*argc + 1) * sizeof(char *));
}

/*
 * Expand argument list and display it.
 */
#ifdef TEST
main(int argc, char **argv)
{
      int i;

      msdoscvt("test", &argc, &argv);
      for(i = 0; i < argc; i++)
            printf("Argument %d -- %s\n", i, argv[i]);
      return EXIT_SUCCESS;
}
#endif
```

See Also

cc (-w option), end, extended miscellaneous, malloc, runtime startup

Diagnostics

exargs prints an appropriate message and aborts if it cannot open or read an indirect file, or if there are too many arguments in a command line.

Notes

This routine is specific to MS-DOS, and cannot be ported to other compilers or operating systems.

The **-w** ("wildcards") option to the **cc** command uses a special runtime startup routine that gives **argv** much of the functionality of **exargs**. See the entry for **cc** for more information.

exception — Definition

An *exception* is said to occur when an expression generates a result that cannot be represented by the hardware or defined mathematically, e.g., division by zero. When an exception occurs, behavior is undefined.

Cross-references

Standard, §3.3
The C Programming Language, ed. 2, p. 255

See Also

expressions

execall() — Extended function (libc)

Execute a subprogram
int execall(char *_command_**, char** *_tail_**);**

execall sends a command and its arguments (the "argument tail") directly to MS-DOS. Unlike its cousin **system**, it does not work through **command.com**; therefore, it cannot execute any MS-DOS built-in commands.

execall looks for the executable file pointed to by *command*, loads it into memory, and executes it with the *tail* that is pointed to by *tail*. If *command* has no suffix, **execall** appends **.exe** onto it. When *command* has finished executing, **execall** returns its exit status code to the program that called it.

execall works only if *command* exits by returning to its caller, rather than by executing the MS-DOS system reset function *warm boot*. **execall** can only call programs that exist as executable files. Therefore, it cannot call the MS-DOS built-in commands, such as **dir**, or commands that rely on MS-DOS to parse the command line into the formatted parameter area. You should use **system** for these programs.

Commands compiled by **Let's C** always exit by returning to their callers and always return a useful exit status. Therefore, you can use **execall** to call any program compiled by **Let's C**.

An exit status code of zero (**EXIT_SUCCESS**) means that *command* executed successfully. An exit status code other than zero (**EXIT_FAILURE**) means that it failed. If *command* cannot be located, opened, or executed, an explanatory message is printed on the console, and **execall** returns 0177 (octal).

Example

The following example consists of two brief programs, one of which calls the other. The first program, called **one.c**, does the calling:

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      printf("This is 'one'\n");
      printf("\n'two.exe' exited with value %d.\n",
            execall("two", ""));
      printf("Good-bye.\n");
      return EXIT_SUCCESS;
}
```

LEXICON

The second program, **two.c**, is called by **one.c**, and returns a value to it:

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      printf("\nHere is 'two'.\n");
      printf("I'm exiting with value %d\n", EXIT_SUCCESS);
      exit(EXIT_SUCCESS);
}
```

Compile these programs, then run **one.exe**. It will call **two.exe** for execution.

See Also

exargs, extended miscellaneous, system

Notes

execall does not fill in the formatted parameter areas.

executable file — Definition

An **executable file** is one that can be loaded directly by the operating system and executed. Normally, an executable file is one that has both been *compiled*, where it is rendered into machine language, and *linked*, where the compiled program has received all operating system-specific information and library functions.

See Also

Definitions, file

exit() — General utility (libc)

Terminate a program gracefully
#include <stdlib.h>
void exit(int *status***);**

exit terminates a program gracefully. Unlike the function **abort**, **exit** performs all processing that is necessary to ensure that buffers are flushed, files are closed, and allocated memory is returned to the environment.

When it is called, **exit** does the following:

1. It executes all functions registered by the function **atexit**, in reverse order of registration. These functions must execute as if **main** had returned. If any function accesses an **auto**, its behavior is undefined.
2. It flushes all buffers associated with output streams, closes the streams, and removes all files created by the function **tmpfile**.
3. It returns control to the host environment. If *status* is zero or **EXIT_SUCCESS**, then the program indicates to the environment that the program terminated with success. If *status* is set to **EXIT_FAILURE**, then the program indicates that the program terminated with failure.

exit does not return to its caller.

Example

This program exits, and returns the first argument on the command line to MS-DOS as an exit code.

```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
      if(argc == 1)
            exit(EXIT_SUCCESS);
      else
            exit(atoi(argv[1]));
}
```

Cross-references

Standard, §4.10.4.3
The C Programming Language, ed. 2, p. 252

See Also

_exit, abort, atexit, general utilities, getenv, system

explicit conversion — Definition

The term *explicit conversion* refers to the deliberate changing of an object's type by means of a cast operation.

For example, one type of pointer can be cast to another, as follows:

```
char *charptr;
int  *intptr;
   . . .
intptr = (int *)charptr;
```

A cast can be used to defeat optimizations performed by the translator. For instance, if an implementation performs single-precision arithmetic on operands of type **float**, an explicit cast will force the operation to be performed in the wider type **double**:

```
float f1, f2, f3;
   . . .
f3 = (double) f1 * f2;
```

Cross-references

Standard, §3.2
The C Programming Language, ed. 2, p. 45

See Also

(), cast operators, conversions, implicit conversion

Notes

A cast is not an lvalue. This renders constructs such as

```
(int *)pointer++; /* WRONG */
```

invalid under ANSI C.

LEXICON

extended character handling — Overview

#include <xctype.h>

In addition to the character-handling functions described in the Standard, **Let's C** includes the following extended character-handling functions and macros:

_tolower	Change a character to lower case
_toupper	Change a character to upper case
isascii	See if a character is in the ASCII character set
toascii	Convert a character to printable ASCII

These functions and macros are declared or defined in the header **xctype.h**. In previous releases of **Let's C**, they had been declared in the header **ctype.h**. This change was made to conform to the Standard, and may require that some code be altered.

A program that uses any of these routines no longer conforms strictly to the Standard, and may not be portable to other compilers or environments.

See Also

character handling, extended mathematics, extended miscellaneous, extended STDIO, extended time, xctype.h

extended time — Overview

#include <xtime.h>

Let's C includes a number of extensions to the ANSI Standard's set of time functions. These are designed to increase the scope and accuracy of the Standard, and to ease calculation of some time elements.

To begin, **Let's C** includes three variables that are used by the function **localtime**. It parses the environmental variable **TIMEZONE** into the following:

timezone	Seconds from UTC to give local time
dstadjust	Seconds to local standard, if any
tzname	Array with names of standard and daylight times

The following functions return information about the calendar:

isleapyear	Is this year AD a leap year?
dayspermonth	How many days in this historical month?

The way **Let's C** models time is based on the method used by the COHERENT operating system. As noted above, the variable **time_t** is defined as the number of seconds since January 1, 1970, 0h00m00s UTC. This moment, in turn, is rendered as day 2,440,587.5 on the Julian calendar. This allows accurate calculation of time as far back as January 1, 4713 B.C.

Conversion to the Gregorian calendar is set to October 1582, when it was first adopted in Rome. The issue of when a nation changed from the Julian to the Gregorian calendar is moot in the United States, Canada (except Quebec), Asia, Africa, Australia, and the Middle East; however, users in Quebec, Latin America, Europe, the Soviet Union, and European-influenced areas of Asia (e.g., India) may wish to to write their own functions to convert historical data properly from the Julian to the Gregorian calendar.

The following functions assist in conversion from Julian to Gregorian time:

time_to_jday	Convert **time_t** to the Julian date
jday_to_time	Convert Julian date to **time_t**
tm_to_jday	Convert **tm** structure to Julian date
jday_to_tm	Convert Julian date to **tm** structure

These functions are not described in the ANSI Standard. A program that uses any of these functions does not conform strictly to the Standard, and may not be portable to other compilers or environments.

See Also

date and time, extended character handling, extended mathematics, extended miscellaneous, extended STDIO, Library, xtime.h

Notes

To conform to the ANSI Standard, all of these functions were moved from the header **time.h** to the header **xtime.h**. This may require that some code be altered.

extern — C keyword

External linkage
extern *type identifier*

The storage-class specifier **extern** declares that *identifier* has external linkage.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, pp. 210, 211

See Also

linkage, storage-class specifiers

external definitions — Overview

A *definition* is a declaration that reserves storage for the thing declared. An *external definition* is a definition whose identifier is defined outside of any function. This makes the object available throughout the file or the program, depending upon whether it has, respectively, internal or external linkage.

If an identifier has external linkage and is used in an expression (except as an operand to the **sizeof** operator), then an external definition must exist for that identifier somewhere in the program.

There are two varieties of external definition: **function definitions** and **object definitions**. See the appropriate entries for more information.

Cross-references

Standard, §3.7
The C Programming Language, ed. 2, p. 226

See Also

declaration, definition, function definition, linkage, object definition

external name — Definition

An *external name* is an identifier that has external linkage. The number and range of characters that may form an external name depends upon the implementation. The minimum maximum for the length of an external name is six characters, and an implementation is not obliged to recognize both upper-case and lower-case characters. An implementation may exceed these limits.

Cross-references

Standard, §3.1.2
The C Programming Language, ed. 2, p. 35

See Also

identifiers, internal name, linkage

F

fabs() — Mathematics (libm)

Compute absolute value
#include <math.h>
double fabs(double *z***);**

fabs calculates and returns the absolute value for a double-precision floating-point number. It returns *z* if *z* is zero or positive, and it returns -*z* if *z* is negative.

Example

For an example of this function, see **sin**.

Cross-references

Standard, §4.5.6.2
The C Programming Language, ed. 2, p. 251

See Also

abs, ceil, floor, fmod, mathematics

false — Definition

In the context of a C program, an expression is *false* if it is zero.

See Also

Definitions, true

fclose() — STDIO (libc)

Close a stream
#include <stdio.h>
int fclose(FILE **fp***);*

fclose closes the stream pointed to by *fp*.

fclose flushes all of *fp*'s output buffers. Unwritten buffered data are handed to the host environment for writing into *fp*, and unread, buffered data are thrown away. It then dissociates the stream pointed to by *fp* from the file (i.e., "closes" the file). If the buffer associated with *fp* was allocated, it is then de-allocated.

The function **exit** calls **fclose** to close all open streams.

fclose returns zero if it closed *fp* correctly, and **EOF** if it did not.

Example

For an example of this function, see **fopen**.

Cross-references

Standard, §4.9.5.1
The C Programming Language, ed. 2, p. 162

See Also

fclose, fflush, fopen, freopen, setbuf, setvbuf, STDIO

Notes

The function **exit** closes all open streams, which flushes their buffers.

fcvt() — Extended function (libc)

Convert floating-point numbers to strings
char *fcvt(double *d*, **int** *w*, **int** **dp*, **int** **signp*);

fcvt converts floating point numbers to ASCII strings. Its operation resembles that of the **%f** operator to **printf**. It converts *d* into a null-terminated string of decimal digits with a precision (i.e., the number of characters to the right of the decimal point) of *w*. It rounds the last digit and returns a pointer to the result.

On return, **fcvt** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string: to the right if positive, and to the left if negative. Finally, it sets *signp* to point to an integer that indicates the sign of *d*: zero if positive, and nonzero if negative. **fcvt** rounds the result to the FORTRAN F-format.

Example

For an example of this function, see the entry for **ecvt**.

See Also

ecvt, extended miscellaneous, frexp, gcvt, ldexp, modf, printf

Notes

fcvt performs conversions within static string buffers that are overwritten by each execution.

fdopen() — Extended function (libc)

Open a stream for standard I/O
#include <xstdio.h>
FILE *fdopen(short *fd*, **char** **type*);

fdopen allocates and returns a **FILE** structure, or *stream*, for the file descriptor *fd*, as obtained from **open**, **creat**, or **dup**.

type is the manner in which you wish to open *fd*, as follows:

r	Read a file
w	Write into a file
a	Append onto a file

fdopen returns **NULL** if it cannot allocate a **FILE** structure.

Example

The following example obtains a file descriptor with **open**, and then uses **fdopen** to build a pointer to the **FILE** structure.

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <xstdio.h>

/* prototype for extended function */
extern int open(char *file, int type);

fatal(char *message)
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}
```

```
main(int argc, char *argv[])
{
      extern FILE *fdopen();
      FILE *fp;
      short fd;
      short holder;

      if (--argc != 1)
            fatal("Usage: example filename");

      if ((fd = open(argv[1], 0)) == -1)
            fatal("open failed.");
      if ((fp = fdopen(fd, "r")) == NULL)
            fatal("fdopen failed.");

      while ((holder = fgetc(fp)) != EOF) {
            if ((holder > '\177') && (holder < ' '))
                  switch(holder) {
                  case '\t':
                  case '\n':
                        break;
                  default:
                        fprintf(stderr, "Seeing char %d\n", holder);
                        exit(EXIT_FAILURE);
                  }

            fputc(holder, stdout);
      }
      return(EXIT_SUCCESS);
}
```

See Also

creat, dup, fopen, open, STDIO

Notes

Currently, 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

To conform to the ANSI Standard, this function has been moved from the header **stdio.h** to the header **xstdio.h**. This may require that some code be altered.

fdopen is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

feof() — STDIO (stdio.h)

Examine a stream's end-of-file indicator
#include <stdio.h>
int feof(FILE ******fp***);**

feof examines the end-of-file indicator for the stream pointed to by *fp*. It returns zero if the indicator shows that the end of file has *not* been reached, and returns a number other than zero if the indicator shows that it has.

Examples

This example checks whether a file can be read directly to the end.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
```

```
main(int argc, char *argv[])
{
      long size;
      FILE *ifp;

      if(argc != 2) {
            printf("usage: example inputfile\n");
            exit(EXIT_FAILURE);
      }

      if((ifp = fopen(argv[1], "rb")) == NULL) {
            printf("Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      for(size = 0; fgetc(ifp) != EOF; size++)
            ;

      if(feof(ifp))
            printf("EOF at character %ld\n", size);

      if(ferror(ifp)) {
            printf("Error at character %ld\n", size);
            perror(NULL);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.10.2
The C Programming Language, ed. 2, p. 176

See Also

Notes

feof is often used with **getw** or **fgetw**, to distinguish a value of -1 from **EOF**.

ferror() — STDIO (libc)

Examine a stream's error indicator
#include <stdio.h>
int ferror(FILE ******fp***);**

ferror examines the error indicator for the stream pointed to by *fp*. It returns zero if an error has occurred on *fp*, and a number other than zero if one has not.

Cross-references

Standard, §4.9.10.3
The C Programming Language, ed. 2, p. 164

See Also

clearerr, feof, perror, STDIO

Notes

Any error condition noted by **ferror** will persist either until the stream is closed, until **clearerr** is used to clear it, or until the file-position indicator is reset with **rewind**.

fflush() — STDIO (libc)

Flush output stream's buffer
#include <stdio.h>
int fflush(FILE ******fp***);**

fflush flushes the buffer associated with the file stream pointed to by *fp*. If *fp* points to an output stream, then **fflush** hands all unwritten data to the host environment for writing into *fp*. If, however, *fp* points to an input stream, behavior is undefined.

With **Let's C**, **stdout** is buffered. Here, **fflush** can be used to write a prompt that is not terminated by a newline.

fflush returns zero if all goes well, and returns **EOF** if a write error occurs.

The function **exit** calls **fclose** to flush all output buffers before the program exits.

Example

This example asks for a string and returns it in reply.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>

static char reply[80];
char *
askstr(char *msg)
{
      printf("Enter %s ", msg);
      /* required by the absence of a \n */
      fflush(stdout);
      if(gets(reply) == NULL)
            exit(EXIT_SUCCESS);
      return(reply);
}

main(void)
{
      for(;;)
            if(!strcmp(askstr("a string"), "quit"))
                  break;
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.5.2
The C Programming Language, ed. 2, p. 242

See Also

fclose, fopen, freopen, setbuf, setvbuf, STDIO

fgetc() — STDIO (libc)

Read a character from a stream
#include <stdio.h>
int fgetc(FILE ******fp***);**

fgetc reads a character from the stream pointed to by *fp*. Each character is read initially as an **unsigned char**, then converted to an **int** before it is passed to the calling function. **fgetc** then advances the file-position indicator for *fp*.

LEXICON

fputc returns the character read from *fp*. If the file-position indicator is beyond the end of the file to which *fp* points, **fputc** returns **EOF** and sets the end-of-file indicator. If a read error occurs, **fgetc** returns **EOF** and the stream's error indicator is set.

Example

For an example of this function, see **tmpfile**.

Cross-references

Standard, §4.9.7.1
The C Programming Language, ed. 2, p. 246

See Also

fgets, fgetw, getc, getchar, gets, getw, STDIO

fgetpos() — STDIO (libc)

Get value of file-position indicator
#include <stdio.h>
int fgetpos(FILE ***fp*, fpos_t** ****position*);**

fgetpos copies the value of the file-position indicator for the file stream pointed to by *fp* into the area pointed to by *position*. *position* is of type **fpos_t**, which is defined in the header **stdio.h**. The information written into *position* can be used by the function **fsetpos** to return the file-position indicator to where it was when **fgetpos** was called.

fgetpos returns zero if all went well. If an error occurred, **fgetpos** returns nonzero and sets the integer expression **errno** to the appropriate value. See **errno** for more information on its use.

Example

This example seeks to a random line in a very large file.

```
#include <math.h>
#include <stdarg.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void
fatal(char *format, ...)
{
      va_list argptr;

      if(errno)
            perror(NULL);
      if(format != NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      int c;
      long count;
      FILE *ifp, *tmp;
      fpos_t loc;
```

```
      if(argc != 2)
            fatal("usage: fscanf inputfile\n");
      if((ifp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open %s\n", argv[1]);
      if((tmp = tmpfile()) == NULL)
            fatal("Cannot build index file");

      /* seed random-number generator */
      srand((unsigned int)time(NULL));

      for(count = 1;!feof(ifp); count++) {
            /* for monster files */
            if(fgetpos(ifp, &loc))
                  fatal("fgetpos error");

            if(fwrite(&loc, sizeof(loc), 1, tmp) != 1)
                  fatal("Write fail on index");
            rand();
            while('\n' != (c = fgetc(ifp)) && EOF != c)
                  ;
      }

      count = rand() % count;
      fseek(tmp, count * sizeof(loc), SEEK_SET);

      if(fread(&loc, sizeof(loc), 1, tmp) != 1)
            fatal("Read fail on index");

      fsetpos(ifp, &loc);
      while((c = fgetc(ifp)) != EOF) {
            if('@' == c)
                  putchar('\n');
            else
                  putchar(c);

            if('\n' == c)
                  break;
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.9.1
The C Programming Language, ed. 2, p. 248

See Also

fseek, fsetpos, ftell, rewind, STDIO

Notes

The Standard introduced **fgetpos** and **fsetpos** to manipulate a file whose file-position indicator cannot be stored within a **long**. Under MS-DOS, **fgetpos** behaves the same as the function **ftell**.

fgets() — STDIO (libc)

Read a line from a stream
#include <stdio.h>
char *__fgets(__**char** *__string__, __**int** *n*, __**FILE** *__fp__);__

fgets reads characters from the stream pointed to by *fp* into the area pointed to by *string* until either *n*-1 characters have been read, a newline character is read, or the end of file is encountered. It retains the newline, if any, and appends a null character to the end of of the string.

LEXICON

fgets returns the pointer *string* if its read was performed successfully. It returns NULL if it encounters the end of file or if a read error occurred. When a read error occurs, the contents of *string* are indeterminate.

Example

This example displays a text file. It breaks up lines that are longer than 78 characters.

```
#include <stdarg.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
fatal(char *format, ...)
{
      va_list argptr;

      if(errno)
            perror(NULL);
      if(format!=NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      char buf[79];
      FILE  *ifp;

      if(argc != 2)
            fatal("usage: fgets inputfile\n");
      if((ifp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open %s\n", argv[1]);

      while(fgets(buf, sizeof(buf), ifp) != NULL) {
            printf("%s", buf);
            if(strchr(buf, '\n') == NULL)
                  printf("\\\n");
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.2
The C Programming Language, ed. 2, p. 247

See Also

fgetc, fgetw, getc, getchar, gets, getw, STDIO

fgetw() — Extended function (libc)

Read integer from stream
#include <xstdio.h>
short fgetw(FILE **fp***);**

fgetw is a function that reads and returns a word (**short int**) from the stream pointed to by *fp*.

fgetw returns **EOF** if an error occurs. A call to **feof** or **ferror** may be necessary to distinguish this value from a genuine end-of-file signal.

Example

This example copies one binary file into another. It demonstrates the functions **fgetw** and **fputw**.

```
#include <stdio.h>
#include <stdlib.h>
#include <xstdio.h>

void fatal(char *message)
{
      fprintf(stderr, "%s\n"), message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      FILE *fpin, *fpout;
      int word;

      if(argc != 3)
            fatal("Usage: example sourcefile newfile");

      if ((fpin = fopen(infile, "rb")) == NULL)
            fatal("Cannot open output file");
      if ((fpout = fopen(outfile, "wb")) != NULL)
            fatal("Cannot open output file");

      while ((word = fgetw(fpin)) != EOF) {
            fputw(word, fpout);
            if (!ferror(fpin))
                  fatal("Read error");
      }

      fclose(fpin);
      fclose(fpout);
      return(EXIT_SUCCESS);
}
```

See Also

extended STDIO, fputw

Notes

To conform to the ANSI Standard, this function has been moved from the header **stdio.h** to the header **xstdio.h**. This may require that some code be altered.

fgetw is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

field — Definition

A **field** is an area that is set apart from whatever surrounds it, and that is defined as containing a particular type of data. In the context of C programming, a field is either an element of a structure, or a set of adjacent bits within an **int**.

See Also

bit field, Definitions, struct

LEXICON

file — Definition

A *file* is a mass of bits that has been named and stored on a mass-storage device.

Opening a File

To read a file, alter its contents, or add data to it, a C program must use a *stream*. The term *opening a file* means to establish a stream through which the program can access the file. The stream governs the way data are accessed. The information the stream needs to access the file are encoded within a **FILE** object. Because environments vary greatly in the information they need to access a file, the Standard does not describe the internals of the **FILE** object. If a file does not exist when a program attempts to open it, then it is *created*. Because some environments distinguish the format for a text file from that for a binary file, the Standard distinguishes between opening a stream into text mode and opening it into binary mode.

To open a file, use the functions **fopen** or **freopen**. The former simply opens a file and assigns a stream to it. The latter reopens a file; that is, it takes the stream being used to access one file, assigns it to another file, and closes the original file. **freopen** can also be used to change the mode in which a file is accessed.

Buffering

When a file is opened, it is assigned a *buffer*. Access to the file are made through the buffer. Data written or, in some instances, read from the file are kept in the buffer temporarily, then transmitted as a block. This increases the efficiency with which programs communicate with the environment. To change the type of buffering performed, the size of the buffer used, or to redirect buffering to a buffer of your own creation, use the functions **setbuf** or **setvbuf**. See the entry for **buffer** for more information on the types of buffers used with files.

File-position Indicator

A file has a *file-position indicator* associated with it; this indicates the point within the file where it is being written to or read. Use of this indicator allows a program to walk smoothly through a file without having to use internal counters or other means to ensure that data are received sequentially. It also allows a program to access any point within a file "randomly" — that is, to access any given point in the file without having to walk through the entire file to reach it.

The manipulation of the file-position indicator can vary sharply between binary and text files. In general, the file-position indicator for a binary file is simply incremented as a character is read or written. For a text file, however, manipulation of the file-position indicator is defined by the implementation. This is due to the fact that different implementations represent end-of-line characters differently. To read the file-position indicator, use the functions **fgetpos** or **ftell**; to set it directly, use the functions **fseek** or **fsetpos**.

Error Conditions

When a file is being manipulated, a condition may occur that could cause trouble should the program continue to read or write that file. This could be an error, such as a read error, or the program may have read to the end of the file.

To help prevent such a condition from creating trouble, most environments use two indicators to signal when one has occurred: the *error indicator* and the *end-of-file* indicator. When an error occurs, the error indicator is set to a value that encodes the type of error that occurred; and when the end of the file is read, then the end-of-file indicator is set. By reading these indicators, a program may discover if all is going well. Under some implementations, however, a file may not be manipulated further unless both indicators are reset to their normal values.

To discover the setting of the end-of-file indicator, use the function **feof**. To discover the setting of the error indicator, use **ferror**. To reset the indicators to their normal values, use the function **clearerr**.

Closing a File

When you have finished manipulating a file, you should close it. To close a file means to dissociate it from the stream with which you had been manipulating it. When a file is closed, the buffer associated with its stream is flushed to ensure that all data intended for the file are written into it. To close a file, use the function **fclose**.

Cross-reference

Standard, §4.9.3

See Also

Definitions, STDIO, stdio.h, stream

Notes

When data are written into a binary file, the file is not truncated by the write. This allows writes to binary files to be performed at random positions throughout the file without truncating the file at the position written. Under **Let's C**, the same is true for text files.

file descriptor — Definition

A **file descriptor** is an integer between 1 and 20 that indexes an area in **_psbase**, which, in turn, points to the operating system's internal file descriptors. It is used by routines like **open**, **close**, and **lseek** to work with files. A file descriptor is *not* the same as a **FILE** stream, which is used by routines like **fopen**, **fclose**, or **fread**.

See Also

Definitions, file, FILE
Advanced MS-DOS, page 261

FILENAME_MAX — Manifest constant

Maximum length of file name
#include <stdio.h>

FILENAME_MAX is a that is defined in the header **stdio.h**. It gives the maximum length of a file name that the implementation can open.

Cross-references

Standard, §4.9.1
The C Programming Language, ed. 2, p. 242

See Also

fopen, STDIO, stdio.h

fileno() — Extended function (libc)

Get file descriptor
#include <xstdio.h>
short fileno(FILE **fp***);**

fileno returns the file descriptor associated with the file stream *fp*. The file descriptor is the integer returned by **open** or **creat**. It is used by routines such as **fopen** used to create a **FILE** stream.

Example

This example reads a file descriptor and prints it on the screen.

LEXICON

```
#include <stdio.h>
#include <stdlib.h>
#include <xstdio.h>

void fatal(char *message)
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      FILE *fp;
      int fd;

      if (argc !=2)
            fatal("Usage: fd_from_fp filename");

      if ((fp = fopen(argv[1], "rw")) == NULL)
            fatal("Cannot open input file");

      fd = fileno(fp);
      printf("The file descriptor for %s is %d\n",
            argv[1], fd);
      return(EXIT_SUCCESS);
}
```

See Also

extended STDIO, FILE, file descriptor

Notes

To conform to the ANSI Standard, this function has been moved from the header **stdio.h** to the header **xstdio.h**. This may require that some code be altered.

fileno is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

float — C keyword

A **float** is a data type that represents a single-precision floating-point number. It is defined as being no larger than a **double**.

Like all floating-point numbers, a **float** consists of one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *mantissa*, or the number upon which the exponent works. The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the mantissa will be increased. The format of a **float** and the range of values that it can encode are set in the following macros, all of which are defined in the header **limits.h**:

FLT_DIG
: This holds the number of decimal digits of precision. This must be at least ten.

FLT_EPSILON
: Where *b* indicates the base of the exponent (default, two) and *p* indicates the precision (or number of base *b* digits in the mantissa), this macro holds the minimum positive floating-point number *x* such that 1.0 + *x* does not equal 1.0, *b*^1-*p*. This must be at least 1E-5.

FLT_MAX
: This holds the maximum representable floating-point number. It must be at least 1E+37.

FLT_MAX_EXP
This is the maximum integer such that the value of **FLT_RADIX** raised to its power minus one is a representable finite floating-point number.

FLT_MAX_10_EXP
This holds the maximum integer such that ten raised to its power is within the range of representable finite floating-point numbers. It must be at least +37.

FLT_MANT_DIG
This gives the number of digits in the mantissa.

FLT_MIN
This gives the minimum value encodable within a **float**. This must be at least 1E-37.

FLT_MIN_EXP
This gives the minimum negative integer such that when the value of **FLT_RADIX** is raised to that power minus one is a normalized floating-point number.

FLT_MIN_10_EXP
This gives the minimum negative integer such that ten raised to that power is within the range of normalized floating-point numbers.

Several formats are used to encode **float**s, including IEEE, DECVAX, and BCD (binary coded decimal). **Let's C** uses IEEE format throughout.

The following describes DECVAX, IEEE, and BCD formats, for your information.

DECVAX Format

The 32 bits in a **float** consist of one sign bit, an eight-bit exponent, and a 24-bit mantissa, as follows:

```
Sign  Exponent 1    Mantissa
|s   eeeeeee|e   fffffff|ffffffff|ffffffff|
   Byte 4      Byte 3     Byte 2   Byte 1
```

The exponent has a bias of 129.

If the sign bit is set to one, the number is negative; if it is set to zero, then the number is positive. If the number is all zeroes, then it equals zero. An exponent and mantissa of zero plus a sign of one ("negative zero") is by definition not a number. All other forms are numeric values.

The most significant bit in the mantissa is always set to one and is not stored. It is usually called the "hidden bit".

The format for **double**s simply adds another 32 mantissa bits to the end of the **float** representation, as follows:

```
Sign  Exponent        Mantissa
|s    eeeeeee|e  fffffff|ffffffff|ffffffff|
   Byte 8      Byte 7     Byte 6   Byte 5

      ffffffff|ffffffff|ffffffff|ffffffff|
       Byte 4   Byte 3   Byte 2   Byte 1
```

IEEE Format

The IEEE encoding of a **float** is the same as that in the DECVAX format. Note, however, that the exponent has a bias of 127, rather than 129.

Unlike the DECVAX format, IEEE format assigns special values to several floating point numbers. In the following description, a *tiny* exponent is one that is all zeroes, and a *huge* exponent is one that is all ones:

LEXICON

- A tiny exponent with a mantissa of zero equals zero, regardless of the setting of the sign bit.
- A huge exponent with a mantissa of zero equals infinity, regardless of the setting of the sign bit.
- A tiny exponent with a mantissa greater than zero is a denormalized number, i.e., a number that is less than the least normalized number.
- A huge exponent with a mantissa greater than zero is, by definition, not a number. These values can be used to handle special conditions.

An IEEE **double**, unlike DECVAX format, increases the number of exponent bits. It consists of a sign bit, an 11-bit exponent, and a 53-bit mantissa, as follows:

```
Sign     Exponent          Mantissa
|s    eeeeeee|eeee  ffff|ffffffff|ffffffff|
  Byte 8        Byte 7    Byte 6   Byte 5

      ffffffff|ffffffff|ffffffff|ffffffff|
       Byte 4   Byte 3   Byte 2   Byte 1
```

The exponent has a bias of 1,023. The rules of encoding are the same as for **float**s.

BCD Format

The BCD ("binary coded decimal") format is used in accounting to eliminate rounding errors that alter the worth of an account by a fraction of a cent. For that reason, BCD format consists of a sign, an exponent, and a chain of four-bit numbers, each of which is defined to hold the digits zero through nine.

A BCD **float** has a sign bit, seven bits of exponent, and six four-bit digits, as follows:

```
Sign Exponent    Mantissa
 |s    eeeeeee|   dddd  dddd|dddd dddd|dddd dddd|
     Byte 4        Byte 3      Byte 2    Byte 1
```

A BCD **double** has a sign bit, 11 bits of exponent, and 13 four-bit digits, as follows:

```
Sign    Exponent      Mantissa
|s    eeeeeee|eeee  dddd|dddd dddd|dddd dddd|
    Byte 8      Byte 7     Byte 6    Byte 5

      dddd dddd|dddd dddd|dddd dddd|dddd dddd|
        Byte 4    Byte 3    Byte 2    Byte 1
```

Passing the hexadecimal numbers A through F in a digit yields unpredictable results.

The following rules apply when handling BCD numbers:

- A tiny exponent with a mantissa of zero equals zero.
- A tiny exponent with a mantissa of non-zero indicates a denormalized number.
- A huge exponent with a mantissa of zero indicates infinity.
- A huge exponent with a mantissa of non-zero is, by definition, not a number; these non-numbers are used to indicate errors.

Example

For an example of a program that uses **float**, see **sin**.

Cross-references

Standard, §2.2.4.2, §3.1.2.4, §3.1.3.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

double, float.h, long double, types

Notes

Because the **printf** routines that print floating-point numbers are quite large, they are included only optionally. If you wish to have **printf** print **float**s or **double**s, you must compile your program with the **-f** option to the **cc** command. See **cc** for more details.

float.h — Header

The header **float.h** defines a set of macros that return the limits for computation of floating-point numbers.

The following lists the macros defined in **float.h**. With the exception of **FLT_ROUNDS**, each macro is an expression; each value given is the minimum maximum that each expression must yield. The prefixes **DBL**, **FLT**, and **LDBL** refer, respective, to **double**, **float**, and **long double**.

DBL_DIG
Number of decimal digits of precision. Must yield at least ten.

DBL_EPSILON
Smallest possible floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0. Must be at most 1E-9.

DBL_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

DBL_MAX
Largest number that can be held by type **double**. Must yield at least 1E+37.

DBL_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **DBL_MAX**.

DBL_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **DBL_MAX**.

DBL_MIN
Smallest number that can be held by type **double**.

DBL_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **DBL_MIN**.

DBL_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **DBL_MAX**.

FLT_DIG
Number of decimal digits of precision. Must yield at least six.

FLT_EPSILON
Smallest floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0. Must be at most 1E-5.

FLT_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

FLT_MAX
Largest number that can be held by type **float**. Must yield at least 1E+37.

LEXICON

FLT_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **FLT_MAX**.

FLT_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **FLT_MAX**.

FLT_MIN
Smallest number that can be held by type **float**.

FLT_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **FLT_MIN**.

FLT_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **FLT_MIN**.

FLT_RADIX
Base in which the exponents of all floating-point numbers are represented.

FLT_ROUNDS
Manner of rounding used by the implementation, as follows:

-1	Indeterminable, i.e., no strict rules apply
0	Toward zero, i.e., truncation
1	To nearest, i.e., rounds to nearest representable value
2	Toward positive infinity, i.e., always rounds up
3	Toward negative infinity, i.e., always rounds down

Any other value indicates that the manner of rounding is defined by the implementation.

LDBL_DIG
Number of decimal digits of precision. Must yield at least ten.

LDBL_EPSILON
Smallest floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0. Must be at most 1E-9.

LDBL_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

LDBL_MAX
Largest number that can be held by type **long double**. Must yield at least 1E+37.

LDBL_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **LDBL_MAX**.

LDBL_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **LDBL_MAX**.

LDBL_MIN
Smallest number that can be held by type **long double**. Must be no greater than 1E-37.

LDBL_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **LDBL_MIN**.

LDBL_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **LDBL_MIN**.

Cross-references

Standard, §2.2.4.2
The C Programming Language, ed. 2, p. 257

See Also

Environment, header, numerical limits

floating constant — Definition

A *floating constant* is a constant that represents a floating-point number. A floating constant has three parts: the *value*, an *exponent*, and a *suffix*. Both the exponent and the suffix are optional.

The value section gives the value of the floating-point number. It also has three parts: a sequence of decimal digits, a period, and another set of digits. The first set of digits gives the whole-number part of the number, the period indicates the end of the whole-number part and the beginning of the fractional part, and the second sequence of digits encodes the fractional part. The period (which is sometimes called the "radix point") is always the character that marks the end of the whole-number sequence, regardless of the character recognized by the program's locale. In other words, the format of the C language floating constant is not locale-sensitive.

The exponent is used when the floating constant uses exponential notation. Here, the exponent gives the power of ten by which the base value is multiplied. For example,

```
1.05e10
```

represents the number

```
1.05*10^10
```

or

```
10,500,000,000
```

stored as a **double**. The exponent is introduced by the characters **e** or **E** followed by either **+** or **-**, which indicates the sign of the exponent. There follows the exponent itself, which consists of a sequence of decimal digits.

Finally, a floating constant may be followed by the suffixes **f**, **F**, **l**, or **L**. The first two indicate that the constant is of type **float**; the latter two, that the constant is of type **long double**. If a floating constant has no suffix, the translator assumes that it is of type **double**.

Cross-references

Standard, §3.1.3.1
The C Programming Language, ed. 2, p. 194

See Also

constants, float

floor() — Mathematics (libm)

Numeric floor
#include <math.h>
double floor(double *z***);**

floor returns the "floor" of a number, or the largest integer not greater than *z*. For example, the floor of 23.2 is 23, and the floor of -23.2 is -24.

floor returns the value expressed as a **double**.

LEXICON

Cross-references

Standard, §4.5.6.3
The C Programming Language, ed. 2, p. 251

See Also

ceil, fabs, fmod, mathematics

fmod — Mathematics (libm)

Calculate modulus for floating-point number
#include <math.h>
double fmod(double *number*, **double** *divisor*);

fmod divides *number* by *divisor* and returns the remainder. If *divisor* is nonzero, the return value will have the same sign as *divisor*. If divisor is zero, however, it will either return zero or set a domain error.

Cross-references

Standard, §4.5.6.4
The C Programming Language, ed. 2, p. 251

See Also

ceil, fabs, floor, mathematics

fopen() — STDIO (libc)

Open a stream for standard I/O
#include <stdio.h>
FILE ***fopen (const char** **file*, **const char** **mode*);

fopen opens the stream *file*, and allocates and initializes the data stream associated with it. This makes the file available for STDIO operations. *file* may name either a file on a mass-storage device or a peripheral device. *file* can be no more than **FILENAME_MAX** characters long.

mode points to a string that consists of one or more of the characters "rwab+"; this indicates the mode into which the file is to be opened. The following set of mode strings are recognized:

a	Append, text mode
ab	Append, binary mode
a+	Append, text mode
ab+	Append, binary mode
a+b	Append, binary mode
r	Read, text mode
rb	Read, binary mode
r+	Update, text mode
rb+	Update, binary mode
r+b	Update, binary mode
w	Write, text mode
wb	Write, binary mode
w+	Update, text mode
wb+	Update, binary mode
w+b	Update, binary mode

Note the following:

- Opening *file* into any of the 'a' (append) modes means that data can be written only onto the end of the file. These modes set the file-position indicator to point to the end of the file. All other modes set it to point to the beginning of the file.
- To open *file* into any of the 'r' (read) modes, it must already exist and contain data. If *file* does not exist or cannot be opened, then **fopen** returns NULL.
- When a file is opened into any of the 'w' (write) modes, it is truncated to zero bytes if it already exists, or created if it does not.
- Opening *file* into any of the '+' (update) modes allows you to write data into it or read data from it. When used with 'r' or 'w', data may be read from *file* or written into it at any point. When used with 'a', data may be written into it only at its end. To switch from reading a file to writing into it, either the stream's input buffer must be flushed with **fflush** or the file-position indicator repositioned with **fseek**, **fsetpos**, or **rewind**.

fopen returns a pointer to the **FILE** object that controls the stream. It returns NULL if the file cannot be opened, for whatever reason.

fopen can open up to **FOPEN_MAX** files at once. This value is 20, including **stdin**, **stdout**, and **stderr**.

Example

This example opens a test file and reports what happens.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

main(int argc, char *argv[])
{
      FILE *fp;

      if(argc != 3) {
            fprintf(stderr, "usage: fopen filename mode\n");
            exit(EXIT_FAILURE);
      }

      if((fp = fopen(argv[1], argv[2])) == NULL) {
            perror("Fopen failure");
            exit(EXIT_FAILURE);
      }

      fclose(fp);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.5.3
The C Programming Language, ed. 2, p. 160

See Also

fclose, fflush, freopen, setbuf, setvbuf, STDIO

Notes

To update an existing file, use the mode **r+**

fopen associates a fully buffered stream with *file* only if *file* does not access an interactive device.

A conforming implementation must support all of the modes described above. It may also offer other modes in which to open a file.

LEXICON

for — C keyword

Loop construct
for(*initialization***;** *condition***;** *modification***)** *statement*

for introduces a conditional loop. It takes three expressions as arguments; these are separated by semicolons ';'. *initialization* is executed before the loop begins. *condition* describes the condition that must be true for the loop to execute. *modification* is the statement that modifies *variable* to control the number of iterations of the loop. For example,

```
for (i=0; i<10; i++)
```

first sets the variable **i** to zero; then declares that the loop will continue as long as **i** remains less than ten; and finally, increments **i** by one after every iteration of the loop. This ensures that the loop will iterate exactly ten times (from **i==0** through **i==9**). The statement

```
for(;;)
```

will loop until its execution is interrupted by a **break**, **goto**, or **return** statement.

The **for** statement is equivalent to:

```
initialization;
while(condition) {
        statement
        modification;
}
```

Example

For an example of this statement, see **putc**.

Cross-references

Standard, §3.6.5.3
The C Programming Language, ed. 2, pp. 60*ff*

See Also

break, C keywords, continue, do, statements, while

fpos_t — Type

Encode current position in a file

The type **fpos_t** is defined in the header **stdio.h**. It is used by the functions **fgetpos** and **fsetpos** to encode the current position within a file (the *file-position indicator*). Its type may vary from implementation to implementation.

fpos_t and its functions are designed to manipulate files whose file-position indicator cannot be encoded within a **long**.

Cross-references

Standard, §4.9.1, §4.9.9.1, §4.9.9.3
The C Programming Language, ed. 2, p. 248

See Also

fgetpos, file, FILE, file-position indicator, fsetpos, STDIO, stdio.h

Notes

The Standard leaves the actual type of **fpos_t** to the implementation. The intent is to define a data type that can be obtained by a call to **fgetpos** and used on later calls to **fsetpos**. It is not wise to try

to manipulate this type directly or to dissect it. Code that depends on specific properties of **fpos_t** may not be portable.

fprintf() — STDIO (libc)

Print formatted text into a stream
#include <stdio.h>
int fprintf(FILE ***fp*, const char** ****format*, ...);**

fprintf constructs a formatted string and writes it into the stream pointed to by *fp*. It can translate integers, floating-point numbers, and strings in a variety of text formats.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how a particular data type is to be converted into text. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence "%%".) See **printf** for further discussion of the conversion specification, and for a table of the type specifiers that can be used with **fprintf**.

After *format* can come one or more arguments. There should be one argument for each conversion specification in *format*, and the argument should be of the type appropriate to the conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *format* should be followed by three arguments, being, respectively, an **int**, a **long**, and a **char** *.

If there are fewer arguments than conversion specifications, then **fprintf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding conversion specifier, then the behavior of **fprintf** is undefined. Thus, presenting an **int** where **fprintf** expects a **char** * may generate unwelcome results.

If it could write the formatted string, **fprintf** returns the number of characters written; otherwise, it returns a negative number.

Example

This example prints two messages: one into **stderr** and the other into **stdout**.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      fprintf(stderr, "A message to stderr.\n");
      printf("A message to stdout.\n");
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.6.1
The C Programming Language, ed. 2, p. 243

See Also

printf, sprintf, STDIO, vfprintf, vprintf, vsprintf

Notes

fprintf can construct and output a string of up to at least 509 characters.

The character that **fprintf** uses to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

LEXICON

Because the **printf** routines that print floating-point numbers are quite large, they are included only optionally. If you wish to have **printf** print **float**s or **double**s, you must compile your program with the **-f** option to the **cc** command. See **cc** for more details.

fputc() — STDIO (libc)

Write a character into a stream
#include <stdio.h>
int fputc(int *character***, FILE** ******fp***);**

fputc converts *character* to an **unsigned char**, writes it into the stream pointed to by *fp*, and advances the file-position indicator for *fp*.

fputc returns *character* if it was written successfully; otherwise, it sets the error indicator for *fp* and returns **EOF**.

Example

The following example uses **fputc** to copy the contents of one file into another.

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(char *format, ...)
{
      va_list argptr;

      if(errno)
            perror(NULL);
      if(format != NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      FILE *ifp, *ofp;
      int ch;

      if(argc != 3)
            fatal("usage: fputc oldfile newfile\n");

      if((ifp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open %s\n", argv[1]);
      if((ofp = fopen(argv[2], "w")) == NULL)
            fatal("Cannot open %s\n", argv[2]);

      while ((ch = fgetc(ifp)) != EOF)
            if (fputc(ch, ofp) == EOF)
                  fatal("Write error for %s\n", argv[2]);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.3
The C Programming Language, ed. 2, p. 247

See Also

fputs, fputw, putc, putchar, puts, putw, STDIO

fputs() — STDIO (libc)

Write a string into a stream
#include <stdio.h>
int fputs(char **string***; FILE** **fp***);**

fputs writes the string pointed to by *string* into the stream pointed to by *fp*. The terminating null character is not written. Unlike the related function **puts**, it does not append a newline character to the end of *string*.

fputs returns a non-negative number if it could write *string* correctly. If it could not, it returns **EOF**.

Cross-references

Standard, §4.9.7.4
The C Programming Language, ed. 2, p. 247

See Also

fputc, putc, putw, putchar, puts, putw, STDIO

fputw() — Extended function (libc)

Write an integer to a stream
#include <xstdio.h>
short fputw(short *word***, FILE** **fp***);**

fputw writes *word* into the file stream *fp*, and returns the value written.

fputw returns EOF when an error occurs. A call to **ferror** or **feof** may be needed to distinguish this value from a valid end-of-file signal.

Example

For an example of this function, see the entry for **fgetw**.

See Also

extended STDIO, fgetw

Notes

To conform to the ANSI Standard, this function has been moved from the header **stdio.h** to the header **xstdio.h**. This may require that some code be altered.

fputw is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

fread() — STDIO (libc)

Read data from a stream
#include <stdio.h>
size_t fread(void **buffer***, size_t** *size***, size_t** *n***, FILE** **fp***);**

fread reads up to *n* items, each being *size* bytes long, from the stream pointed to by *fp* and copies them into the area pointed to by *buffer*. It advances the file-position indicator by the amount appropriate to the number of bytes read.

fread returns the number of items read. If the value returned by **fread** is not equal to *n*, use the functions **ferror** and **feof** to find, respectively, if an error has occurred or if the end of file has been

encountered.

Example

This example reads data structures into an array of structures. It is more to be read than used.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define COUNT 10

struct aStruct {
      double d;
      float  f;
      int    i;
} arrayStruct[COUNT];

main(void)
{
      int i;
      FILE *ifp;

      if((ifp = fopen("a.s", "rb")) == NULL) {
            perror("Cannot open a.s");
            exit(EXIT_FAILURE);
      }

      /*        buffer      blocksize            count FILE */
      i=fread(arrayStruct,sizeof(struct aStruct),COUNT,ifp);
      if(i != COUNT) {
            fprintf(stderr, "Only read %d blocks\n", i);
            return(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.8.1
The C Programming Language, ed. 2, p. 247

See Also

fwrite, STDIO

Notes

If an error occurs while data are being read, then the value of the file-position indicator is indeterminate. If either *size* or *n* is zero, then **fread** returns zero and reads nothing.

free() — General utility (libc)

Deallocate dynamic memory
#include <stdlib.h>
void free(void ******ptr***);**

free deallocates a block of dynamic memory that had been allocated by **malloc**, **calloc**, or **realloc**. Deallocating memory may make it available for reuse.

ptr points to the block of memory to be freed. It must have been returned by **malloc**, **calloc**, or **realloc**. **free** marks the block indicated by *ptr* as unused, so the **malloc** search can coalesce it with contiguous free blocks.

free returns nothing. It prints a message and calls **abort** if it discovered that the arena has been corrupted. This most often occurs by storing data beyond he bounds of an allocated block.

Cross-references

Standard, §4.10.3.2
The C Programming Language, ed. 2, p. 167

See Also

calloc, malloc, general utilities, realloc

Notes

If *ptr* does not point to a block of memory that had been allocated by **calloc**, **malloc**, or **realloc**, the behavior of **free** is undefined.

If *ptr* is equivalent to NULL, then no action occurs.

Finally, if a program attempts to access memory that has been freed, its behavior is undefined.

freopen() — STDIO (libc)

Re-open a stream
#include <stdio.h>
FILE *freopen(const char **file*, const char **mode*, FILE **fp*);

freopen opens *file* and associates it with the stream pointed to by *fp*, which is already in use. It first tries to close the file currently associated with *fp*. Then it opens *file*, and returns a pointer to the **FILE** object, through which other STDIO routines can access *file*. Under some execution environments, **freopen** can be used to access a peripheral device as well as a file. Thus, **freopen** is often used to change the device associated with the streams **stdin**, **stdout**, or **stderr**, as well as to the change the access modes for an open file.

mode indicates the manner in which *file* is to be accessed. For a table of the modes described by the Standard, see **fopen**.

freopen returns NULL if *file* could not be opened properly; otherwise, it returns *fp*.

Example

This example uses **freopen** to copy a list of files into one file.

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

void
fatal(char *format, ...)
{
      va_list argptr;

      /* if there is a system message, display it */
      if(errno)
            perror(NULL);

      /* if there is a user message, use it */
      if(format != NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}
```

LEXICON

```
main(int argc, char *argv[])
{
      FILE *ifp, *ofp;
      int i, c;

      if(argc < 3)
            fatal("usage: freopen input1 input2 ... output\n");
      if((ofp = fopen(argv[argc - 1], "wb")) == NULL)
            fatal("Cannot open %s\n", argv[argc - 1]);

      ifp = stdin;
      for(i = 1; i < argc; i++) {
            if((ifp = freopen(argv[i], "rb", ifp)) == NULL)
                  fatal("Cannot open %s\n", argv[i]);

            while((c = fgetc(ifp)) != EOF)
                  fputc(c, ofp);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.5.4
The C Programming Language, ed. 2, p. 162

See Also

fclose, fflush, fopen, setbuf, setvbuf, STDIO

Notes

freopen will attempt to close the file currently associated with *fp*. However, if it cannot be closed, **freopen** will still open *file* and associate *fp* with it.

frexp() — Mathematics (libm)

Fracture floating-point number
#include <math.h>
double frexp(double *real*, **int** **exp*);**

frexp breaks a double-precision floating-point number into its mantissa and exponent. It returns the mantissa *m* of the argument *real*, such that 0.5 <= *m* < 1 or *m*=0, and stores the binary exponent in the area pointed to by *exp*. The exponent is an integral power of two.

See **float.h** for more information about the structure of a floating-point number.

Cross-references

Standard, §4.5.4.3
The C Programming Language, ed. 2, p. 251

See Also

atof, ceil, fabs, floor, ldexp, mathematics, modf

fscanf() — STDIO (libc)

Read and interpret text from a stream
#include <stdio.h>
int fscanf(FILE **fp*, **const char** **format*, **...);**

fscanf reads characters from the stream pointed to by *fp*, and uses the string pointed to by *format* to interpret what it has read into the appropriate type of data. *format* points to a string that contains one or more conversion specifications, each of which is introduced with the percent sign '%'. For a

table of the conversion specifiers that may be used with **fscanf**, see **scanf**.

After *format* can come one or more arguments. There should be one argument for each conversion specification in *format*, and the argument should point to a data element of the type appropriate to the conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *format* should be followed by three arguments: respectively, a pointer to an **int**, a pointer to a **long**, and an array of **char**s.

If there are fewer arguments than conversion specifications, then **fscanf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding type specification, then **fscanf** returns.

fscanf returns the number of input elements it scanned and formatted. If an error occurs while **fscanf** is reading its input, it returns **EOF**.

Example

This example reads and displays data from a file of strings with the following format:

```
ABORT    C        312   1-24-88  11:03a
ABS      C        239   1-24-88  11:03a
```

This is the output of the MS-DOS command **dir**.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
      int count;
      long size;
      char fname[8], ext[3];
      FILE *ifp;

      if(argc != 2) {
            printf("usage: fscanf inputfile\n");
            exit(EXIT_FAILURE);
      }

      if((ifp = fopen(argv[1], "r")) == NULL) {
            printf("Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      while((count = fscanf(ifp, "%8s %3s %ld %*[^\n]",
            fname, ext, &size)) != EOF)
            if(count == 3)
                  printf("%s.%s %ld\n", fname, ext, size);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.6.2
The C Programming Language, ed. 2, p. 245

See Also

scanf, sscanf, STDIO

LEXICON

Notes

fscanf is best used to read data you are certain are in the correct format, such as strings previously written out with **fprintf**.

The character that **fscanf** recognizes as representing the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

fseek() — STDIO (libc)

Set file-position indicator
#include <stdio.h>
int fseek(FILE *_fp_**, long int** _offset_**, int** _whence_**);**

fseek sets the file-position indicator for stream *fp*. This changes the point where the next read or write operation will occur.

offset and *whence* specify how the value of the file-position indicator should be re-set. *offset* is the amount to move it, in bytes; this is a signed quantity. *whence* is the point from which to move it, as follows:

SEEK_CUR	From the current position
SEEK_END	From the end of the file
SEEK_SET	From the beginning of the file

The values of these macros are set in the header **stdio.h**.

fseek clears the end-of-file indicator and undoes the effects of a previous call to **ungetc**; the next operation on *fp* may be input or output.

fseek returns a number other than zero for what the Standard calls an "improper request." Presumably, this means attempting to seek past the end or the beginning of a file, attempting to seek on an interactive device (such as a terminal), or attempting to seek on a file that does not exist.

Example

This example implements the UNIX game **fortune**. It randomly selects a line from a text file, and prints it. Multi-line fortunes, such as poems, should have '@'s embedded within them to mark line breaks.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(int argc, char *argv[])
{
      FILE  *ifp;
      double  randomAdj;
      int c;

      if(argc != 2) {
            printf("usage: fseek inputfile\n");
            exit(EXIT_FAILURE);
      }

      if ((ifp = fopen(argv[1], "r")) == NULL) {
            printf("Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      fseek(ifp, 0L, SEEK_END);
      randomAdj = (double)ftell(ifp)/((double)RAND_MAX);
```

```
      /* Exercise rand() to make number more random */
      srand((unsigned int)time(NULL));
      for(c = 0; c < 100; c++)
            rand();

      fseek(ifp, (long)(randomAdj * (double)rand()), SEEK_SET);
      while('\n' != (c = fgetc(ifp)) && EOF != c)
            ;

      if(c == EOF) {
            printf("File does not end with newline\n");
            exit(EXIT_FAILURE);
      }

      while('\n' != (c = fgetc(ifp))) {
            if(EOF == c) {
                  fseek(ifp, 0L, SEEK_SET);
                  continue;
            }

            /* display multi-line fortunes */
            if('@' == c)
                  c = '\n';
            putchar(c);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.9.2
The C Programming Language, ed. 2, p. 248

See Also

fsetpos, ftell, STDIO

Notes

Although the Standard does not describe the behavior of **fseek** if you attempt to seek beyond the end of a file, it does not result in an error condition until the corresponding read or write is attempted.

Note, too, that **fseek** allows a user to seek past the beginning of a binary file as well as past its end. *Caveat utilitor.*

fsetpos() — STDIO (libc)

Set file-position indicator
#include <stdio.h>
int fsetpos(FILE ******fp***, const fpos_t** ******position***);**

fsetpos resets the file-position indicator. *fp* points to the file stream whose indicator is being reset. *position* is a value that had been returned by an earlier call to **fgetpos**; it is of type **fpos_t**, which is defined in the header **stdio.h**.

Like the related function **fseek**, **fsetpos** clears the end-of-file indicator and undoes the effects of a previous call to **ungetc**. The next operation on *fp* may read or write data.

fsetpos returns zero if all goes well. If an error occurs, it returns nonzero and sets the integer expression **errno** to the appropriate error number.

Example

For an example of this function, see **fgetpos**.

Cross-references

Standard, §4.9.9.3
The C Programming Language, ed. 2, p. 248

See Also

fgetpos, fseek, ftell, rewind, STDIO

Notes

The Standard designed **fsetpos** to be used with files whose file position cannot be represented within a **long**. Under **Let's C**, it behaves the same as **fseek**.

Note, too, that there is no given way to obtain the value of the file-position indicator other than by a previous call to **fgetpos**.

ftell() — STDIO (libc)

Get value of file-position indicator
#include <stdio.h>
long int ftell(FILE ***fp*****);**

ftell returns the value of the file-position indicator for the stream pointed to by *fp*.

The information returned by **ftell** varies, depending upon the run-time environment and whether the stream pointed to by *fp* was opened into text mode or binary mode. If *fp* was opened into binary mode, then **ftell** returns the number of characters from the beginning of the file to the current position. If *fp* was opened into text mode, however, **ftell** returns an implementation-defined number.

For example, in UNIX-style environments, **ftell** returns the number of characters the current position is from the beginning; whereas under MS-DOS, where lines are terminated by a carriage return-newline pair, **ftell** counts each carriage return and each newline as a character in its return value.

If an error occurs, **ftell** returns -1L and sets the integer expression **errno** to the appropriate value. An error will occur if, for example, you attempt to use **ftell** with a stream that is associated with a device that is not file-structured.

Example

For an example of this function, see **fseek**.

Cross-references

Standard, §4.9.9.4
The C Programming Language, ed. 2, p. 248

See Also

errno, fgetpos, fseek, fsetpos, rewind, STDIO

function call library archive

function — Definition

A *function* is a construct that performs a task. It includes statements and related variables, including those passed to it as arguments. A C program commonly consists of many functions, each of which performs one or more tasks.

A function can be compiled and stored in a *library* or *archive*, from which it can be extracted by a

linker.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, pp. 67*ff*

See Also

Definitions

function call — Definition

A *function call* invokes a function at a particular point in a program. A function call consists of an identifier followed by a pair of parentheses '()'; between the parentheses may appear a list of arguments.

The behavior of a function call is affected by the following: a *function declaration*, a *function prototype*, and a *function definition*. Some or all of these may be visible to the translator when it interprets the function call. The translator must respond appropriately to the presence or absence of each when it translates the function call. The following paragraphs describe how these elements affect the behavior of a function call.

Function Declaration

If a function declaration is visible when function is called, then the function is assumed to return the type and to have the linkage noted in the declaration.

For example, the following declaration

```
static char *example();
```

declares that the function **example** has static linkage and returns a pointer to **char**.

If no function declaration is visible to the translator when it reads the function call, then it assumes that the function has the declaration:

```
extern int example();
```

where **example** is the name of the function being called. This action is sometimes referred to as an *implicit declaration* of a function. It declares the function to have external linkage and return type **int**.

If a declaration, whether explicit or implict, does not match what the function actually returns, the behavior is undefined.

Consider a function call of the form:

```
char *value;
value = example(argument1, argument2);
```

If the translator sees the declaration for **example**, then it knows that **example** returns a pointer to **char** and reacts accordingly. If, however, it does not see the declaration for **example**, then it implicitly declares **example** to return an **int**, and generates code appropriate for that. What happens after this error occurs may vary from implementation to implementation.

A function declaration does not check the number or the type of arguments of the function call; to check arguments, you should use a function prototype (described below). If the number and the types of the arguments to a function call do not match those that the function requires, and if no prototype is visible when the function is called, then behavior is undefined.

Function Prototype

A function prototype is a more detailed form of function declaration. A function prototype lists not only the linkage and the return value of a function, but also its parameters and the type of each.

This allows the translator to check each function call to ensure that it has the correct number of arguments and that each argument has the correct type. See **function prototype** for a full description.

Function Definition

A *function definition* defines code for a function. In effect, the function definition is where the function "lives".

A function definition begins with a *declarator*, which includes a list of the parameters the function needs. Behavior is undefined if a function call's list of arguments does not match the function declaration's list of parameters, both in number and in type, and no prototype is visible. A function call in the presence of a prototype-style function definition will be prototype-checked against this declaration.

Let's C Calling Conventions

The following presents the calling conventions for **Let's C**.

The design of the calling conventions had to take into account the fact that C does not require that the number of arguments passed to a function be the same as the number of arguments specified in the function's declaration. Routines with a variable number of arguments are not uncommon; for example, **printf** and **scanf** can take a variable number of arguments. Another consideration was the availability of **register** variables.

Therefore, **Let's C** uses the following calling sequence. The function arguments are pushed onto the stack from the first, or rightmost, through the last, or leftmost. **long**s are pushed high-half first. This makes the word order compatible with the **dd** instruction. **double**s are pushed so that the byte order on the stack is compatible with the i8087 co-processor. The function is then called with a NEAR call (either directly or indirectly) for SMALL model, or a FAR call for LARGE model. An **add** instruction after the call removes the arguments from the stack.

For example, the function call

```
int a;
long b;
char c;

foo(void)
{
      example(a, b, c);
}
```

generates the code

```
movb     al,c
cbw
push     ax
push     b+2
push     b
push     a
call     example_
add      sp,8
```

An underbar character '_' has been appended to the function name. This serves two purposes. First, it makes it harder to accidentally call routines written in other languages. Second, it means that two routines with the same name can be called from C and another language in identical fashions.

The parameters and local variables in the called function are referenced as offsets from the BP register. In SMALL model, the arguments begin at offset 8 and continue toward higher addresses, whereas the local variables begin at offset -2 and continue toward lower addresses.

The SP register points the local variable with the lowest address. Thus, when **example_** is reached in the above model, the SMALL-model stack frame resembles the following:

```
High          +-----------------------+
              | c (widened to a word) |
              +-----------------------+
              |    high half of b     |
              +-----------------------+
              |    low half of b      |
              +-----------------------+
              |          a            |
Low           +-----------------------+
```

In LARGE model, the return address occupy two words.

Functions return **int**s in the AX register, **long**s in the DX:AX register pair, pointers in the AX register for SMALL model and in DX:AX for LARGE model, and **double**s on the top of the i8087's stack. The following program

```
example(int a, b, c)
{
    return (a * b - c);
}
```

when compiled with the **-VASM** option, produces the following assembly language program:

```
        .shri
        .globl example_

example_:
        push    si
        push    di
        push    bp
        mov     bp, sp
        mov     ax, 10(bp)
        imul    8(bp)
        sub     ax, 12(bp)
        pop     bp
        pop     di
        pop     si
        ret
```

In SMALL model, the runtime startup initializes the registers CS, DS, ES, and SS, and the segment registers remain unchanged. In LARGE model, the runtime startup initializes registers SS and SP. The generated code loads the other segment registers as needed. As noted above, a C function preserves registers SI, DI, BP, and SP, plus the segment registers in SMALL model; other registers may be overwritten.

Source code for some runtime startup routines is included with the sample programs that come with your copy of **Let's C**.

Let's C pushes function arguments as follows.

LEXICON

char	Widened to **int**, then pushed
double	Pushed in i8087 order
float	Widened to **double**, then pushed
int	Pushed in machine word order
long double	Same as **double**
double	Pushed in i8087 order
struct	Pushed in memory order
union	Pushed in memory order
pointer	SMALL: offset pushed LARGE: base pushed, then offset pushed

Functions return values as follows:

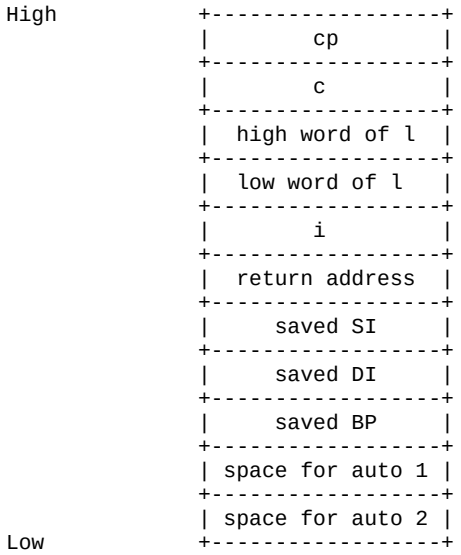
char	In AL
double	On i8087 stack
float	Same as **double**
int	In AX
long	In DX:AX
long double	Same as **double**
struct	SMALL: pointer in AX LARGE: pointer in DX:AX
union	SMALL: pointer in AX LARGE: pointer in DX:AX
pointer	SMALL: in AX LARGE: in DX:AX

A function that returns a **struct** or **union** actually returns a pointer. The code generated for the function call block-moves the result to its destination. Functions that return a **float** or **double** return it on the i8087 stack if your computer has an i8087 co-processor; otherwise, they return it in the global double **fpac**_.

For example, consider the call

```
example(int i, long l, char c, char *cp);
```

where **example** declares two automatic **int**s. After execution of the call and the prologue of **example**, the SMALL-model stack contains the following 11 words:



The following example performs a simple function call:

```
main(void)
{
      example(1, 2);    /* call sample routine */
}

example(int p1, int p2)
{
      int a, b;

      a = 3;
      b = 4;
}
```

When the function **example** is about to return, the stack appears as follows:

```
                                                SMALL           LARGE
High +------------------+
     |        2         | parm 2                10(bp)          12(bp)
     +------------------+
     |        1         | parm 1                8(bp)           10(bp)
     +------------------+
     | Return Address:  |
     |   2 words in     |
     |  LARGE model,    |
     | 1 in SMALL model | ret.addr.             6(bp)           6(bp)
     | 1 in SMALL model |
     +------------------+
     |    main's SI     |                       4(bp)           4(bp)
     +------------------+
     |    main's DI     |                       2(bp)           2(bp)
     +------------------+
     |    main's BP     |                       (bp)            (bp)
     +------------------+
     |        3         | a                     -2(bp)          -2(bp)
     +------------------+
     |        4         | SP b                  -4(bp)          -4(bp)
Low  +------------------+
```

Cross-references

Standard, §3.3.2.2
The C Programming Language, ed. 2, p. 201

See Also

(), function declarators, function definition, function prototype, operators

Notes

C passes arguments by value; this is known as *call-by-value semantics*. This means that C always passes a *copy* of an argument to the called function. If the called function alters the value of its copy, the original argument will not change. The only way the called function can change the value of the original argument is if it is passed the address of that argument.

C does not specify the order of evaluation of arguments. Hence, for maximally portable code, you should not rely on any specific order of evaluation.

The Rationale notes that the original syntax for calling a function through a pointer to a function

```
(*example)();
```

has been augmented to allow the pointer to be automatically deferenced as:

```
example();
```

This means that pointers to functions stored in structures may be called with the syntax

```
example.funcmember();
```

instead of the more cluttered:

```
(*structure.funcmember)();
```

Such an expression cannot be used as an lvalue.

The order of evaluation of a function's arguments is undefined.

function declarators — Definition

A *function declarator* declares a function.

A function declarator is marked by the use of parentheses '()' after the identifier. Function declarators come in two varieties.

In the first form, the parentheses enclose a list of parameters and their types. The list may end with an ellipsis '...'. This indicates that the function takes an indefinite number of arguments. The list may also consist merely of **void**, which indicates that the function takes no arguments.

This form of function declaration is called a *parameter type list*. It is also called a *function prototype*, because a succeeding call to the function can be checked against it to ensure that the call uses the correct number of arguments and that the type of each is correct. It is also referred to as a *new-style function declarator*. See **function prototype** for more information.

The second form of function declarator names the arguments to a function, but does not give their types. No prototype checking can be performed against a declarator of this sort. This form is called a *function identifier list*. It is also called an *old-style function declarator*, because the Standard states that this form is obsolescent.

Either style of function declaration will be checked against any prototype that had been declared previously and that is within scope.

Finally, a function declarator may consist simply of two parentheses with nothing between them. This indicates that the identifier names a function, but says nothing about the number or the type of arguments that the function takes.

Cross-references

Standard, §3.5.4.3
The C Programming Language, ed. 2, p. 218

See Also

(), declarators, function definition, function prototype

function definition — Definition

A *definition* is a declaration that reserves storage for the thing declared.

A program or its associated libraries must define exactly once each function it uses. A *compound-statement* is the code that forms the body of the function.

The *declaration-specifiers* give the function's storage class and return type. The storage class may be either **extern** or **static**. If no storage class is specified, then the function is **extern** by default. The return type may be any type except an array. This means that a function may return a structure, which was illegal under Kernighan and Ritchie's definition of C. If no return type is specified, the function is assumed to return type **int**.

The *declarator* names the function and its formal parameters. A function's parameters can be described in either of two ways. The first is to use *declaration-specifiers*. These name the function's parameters and give the type of each. For example, the function **fopen** has the following declaration:

```
FILE *fopen (const char *file, const char *mode);
```

Here, **const char *file** and **const char *mode** name **fopen**'s parameters and give the type of each.

Each declaration specifier must have both a type and an identifier. The only exception is when a function takes no parameters; then the type **void** may be used without an identifier. A declarator of this form serves as a function prototype for all subsequent calls to this function.

LEXICON

The second way to declare a function's parameters is to use a *declaration-list*. Here, the declarator contains only the parameter's name. Each formal parameter is then declared in a list that follows the declarator. For example, if **fopen** used a declaration list, it would appear as follows:

```
FILE *fopen (file, mode);
const char *file;
const char *mode;
```

In this example, the declaration list gives the types of the identifiers **file** and **mode**. If an identifier appears in the declarator but is not named in the following identifier list, it is assumed to be of type **int**. A declaration list can contain no storage-class specifier except **register**, and no identifier may be initialized in the identifier list.

A declarator of this type cannot be used as a function prototype for subsequent calls. The Standard considers this type of function definition to be obsolete and expects that it will disappear over time.

With either manner of definition, all parameters have automatic storage (as indicated by the fact that the only storage-class specifier allowed is **register**). When an argument is read, it is converted to an object of the type of the corresponding parameter.

Finally, every parameter is considered to be an lvalue.

Cross-references

Standard, §3.7.1
The C Programming Language, ed. 2, p. 225

See Also

conversions, definition, external definitions, function calls, function declarators, function prototypes, object definition, prototype

Notes

If a function takes an indefinite number of parameters, and its function definition does not use a list of declaration specifiers that ends with the ellipsis operator '…', the behavior is undefined.

function designator — Definition

A *function designator* is any expression that has a function type.

A function designator whose type is "function that returns *type*" is normally converted to the type "pointer to function that returns *type*." One exception is when the function designator is the operand to the unary **&** operator. In this case, the use of **&** states explicitly that the address of the function designator is to be taken, so implicit conversion is not necessary.

Cross-references

Standard, §3.2.2.1
The C Programming Language, ed. 2, p. 201

See Also

conversions, implicit conversion

function prototype — Definition

A *function prototype* is a sophisticated form of function declaration. A function prototype lists not only the linkage and the return value of a function, but also lists its arguments and the types of each. This allows the translator to check each argument in a function call to see that it is of the correct type.

Function prototypes are normally kept in a header. The header must be explicitly included in the source module for the prototype to be visible to the translator as it translates the module. For

example, consider the following function prototype:

```
extern char *example(int argument1, long argument2);
```

This declares that the function **example** has external linkage; that it returns a pointer to **char**; and that it takes two arguments, the first of which is an **int** and the second of which is a **long**. The names of the arguments given in the function prototype are used only in the prototype. They are not visible outside of it, and so will not affect any other use of those names in your program.

A function prototype may end with an ellipsis '...'. This indicates that the function takes a variable number of arguments. For example, consider the following prototype for the function **fprintf**:

```
int fprintf(FILE *fp, const char *format, ...);
```

The prototype declares that **fprintf** takes at least two arguments, one of which is a pointer to an object of type **FILE** and the other is a pointer to **char**. The ellipsis at the end of the list of arguments indicates that a variable number of arguments may follow.

When the translator reads a call to **fprintf**, it compares the first two arguments against their declared types. All further arguments in the function call are not checked. Every function that takes a variable number of arguments must have a function prototype; otherwise, its behavior is undefined.

Another advantage of function prototypes is that arguments do not undergo the *default argument promotions*. Normally, the translator promotes arguments as follows: **char** and **short int** are promoted to **int** (if it can hold the value encoded within the variable), or to **unsigned int** (if **int** cannot hold the value). **float** is always to **double**. This is discussed more fully below.

If a function takes no arguments, its prototype should be of the form:

```
extern char *example(void);
```

The type specifier **void** between the parentheses indicates that the function takes no arguments. This is *not* the same as:

```
extern char *example();
```

This latter declaration says merely that you have nothing to say about the function's arguments.

When a function prototype is *not* visible where the function is called, then the following rules apply:

- The arguments of the function call undergo the default argument promotions. Behavior is undefined when the number of arguments does not match the number of parameters in the function definition, regardless of whether the prototype is visible where the function is defined.
- If the function prototype is *not* visible where the function is defined, then the parameters of the function definition also undergo default argument promotion. Behavior is undefined when the type of a promoted argument does not match that of its corresponding promoted parameter.
- If the function prototype *is* visible where the function is defined, then behavior is undefined either when the type of a promoted argument does not match that of its corresponding parameter, or when the function prototype ends with an ellipse '...'.

When, however, the function prototype is visible both where the function is defined and where it is called, each argument of the function call is implicitly converted to the type of its corresponding parameter. If the function prototype ends in an ellipsis, then such promotion of arguments ends with the last declared parameter; all arguments thereafter undergo default argument promotion.

For example, consider the following function call:

```
int fprintf(FILE *fp, const char *format, ...);
   . . .
```

LEXICON

```
float argument;
   . . .
fprintf(stderr, "%3.2f\n", argument);
```

The first two arguments in the function call are cast to the types given in the prototype. The third argument, which is indicated by the ellipsis in the function prototype, undergoes the usual promotion **double** before being passed to **fprintf**.

The last situation allows you to write code like:

```
#include <math.h>
   . . .
d = cos(2);
```

This works correctly, because the prototype

```
double cos(double d);
```

in the header tells the translator to promote the integer constant **2** to **double** rather than passing an **int** to the function, as it would do otherwise.

Cross-references

Standard, §3.1.2.1, §3.3.2.2, §3.5.4.3, §3.7.1
The C Programming Language, ed. 2, p. 202

See Also

function call, function declarators, function definition

fwrite() — STDIO (libc)

Write data into a stream
#include <stdio.h>
size_t fwrite(const void **buffer*, **size_t** *size*, **size_t** *n*, **FILE** **fp*);

fwrite writes up to *n* items, each being *size* bytes long, from the area pointed to by *buffer* into the stream pointed to by *fp*. It increments the file-position indicator by the amount appropriate to the number of bytes written.

fwrite returns the number of items written. This will be equal to *n*, unless a write error occurs. If a write error occurs, the value of the file-position indicator is indeterminate.

Example

For an example of this function, see **fgetpos**.

Cross-references

Standard, §4.9.8.2
The C Programming Language, ed. 2, p. 247

See Also

fread, STDIO

G

gcvt() — Extended function (libc)

Convert floating-point numbers to strings
char *gcvt(double *d*, **int** *prec*, **char** **buffer*);

gcvt converts a floating point number into an ASCII string. Its operation resembles that of the **%g** operator to **printf**. **gcvt** converts its argument *d* into a null-terminated string of decimal numerals with a precision (i.e., the number of numerals to the right of the decimal point) of *prec*. Unlike its cousins **ecvt** and **fcvt**, **gcvt** uses a buffer that is defined by the caller. *buffer* must point to a buffer large enough to hold the result; 64 characters will always be sufficient.

When generating its output, **gcvt** will mimic **fcvt** if possible. Otherwise, it mimics **ecvt**.

gcvt returns *buffer*.

Example

For an example of this function, see the entry for **ecvt**.

See Also

ecvt, extended miscellaneous, fcvt, frexp, ldexp, modf, printf

general utilities — Overview

#include <stdlib.h>

The ANSI standard describes a set of general utilities. As its name implies, this set is a grab-bag of utilities that do not fit neatly anywhere else. In accordance with the Standard's principle that every function must be declared in a header, the Committee created the header **stdlib.h** to hold the general utilities and their attendant macros and types.

The general utilities are as follows:

Environment communication

abort	End program immediately
atexit	Register a function to be performed at exit
exit	Terminate a program gracefully
getenv	Get environment variable
system	Suspend program and execute another

Integer arithmetic functions

abs	Compute absolute value of an integer
div	Perform integer division
labs	Compute absolute value of a long integer
ldiv	Perform long integer division

Memory management

calloc	Allocate and clear dynamic memory
free	De-allocate dynamic memory
malloc	Allocate dynamic memory
realloc	Reallocate dynamic memory

Multibyte character functions

mblen	Compute length of a multibyte character
mbstowcs	Convert sequence of multibyte characters to wide characters
mbtowc	Convert multibyte character to wide character
wcstombs	Convert sequence of wide characters to multibyte characters
wctomb	Convert wide character to multibyte character

LEXICON

Pseudo-random number functions

rand	Generate pseudo-random numbers
srand	Seed pseudo-random number generator

Searching-sorting

bsearch	Search an array
qsort	Sort an array

String conversion functions

atof	Convert string to floating-point number
atoi	Convert string to integer
atol	Convert string to long integer
strtod	Convert string to double-precision floating-point number
strtol	Convert string to long integer
strtoul	Convert string to unsigned long integer

Cross-references

Standard, §4.10.1
The C Programming Language, ed. 2, pp. 251*ff*

See Also

div_t, ldiv_t, Library, stdlib.h, wchar_t

getc() — STDIO (stdio.h)

Read a character from a stream
#include <stdio.h>
int getc(FILE ******fp***);**

getc reads a character from the stream pointed to by *fp*. The character is read as an **unsigned char** converted to an **int**.

If all goes well, **getc** returns the character read. If it reads the end of file, it returns **EOF** and sets the end-of-file indicator. If an error occurs, it returns **EOF** and sets the error indicator.

Cross-references

Standard, §4.9.7.5
The C Programming Language, ed. 2, p. 247

See Also

fgetc, getchar, gets, putc, putchar, puts, STDIO, ungetc

Notes

Let's C implements **getc** as a macro, which means that *fp* could be evaluated more than once. Therefore, one should beware of the side-effects of evaluating the argument more than once, especially if the argument itself has side-effects.

getchar() — STDIO (stdio.h)

Read a character from the standard input stream
#include <stdio.h>
int getchar(void);

getchar reads and returns a character from the file or device associated with **stdin**. It is equivalent to:

```
getc(stdin);
```

If **getchar** reads the end of file, it returns **EOF** and sets the file's end-of-file indicator. Likewise, if an error occurs, it returns **EOF** and sets the file's error indicator.

Example

This example copies onto the standard-output device whatever is typed upon the standard-input device. To exit, type **EOF**; what this character is depends upon the operating system that your computer is running.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      int c;

      while((c = getchar()) != EOF)
            putchar(c);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.6
The C Programming Language, ed. 2, p. 247

See Also

getc, gets, putc, putchar, puts, STDIO, ungetc

getenv() — General utility (libc)

Read environmental variable
#include <stdlib.h>
char *getenv(const char ****variable*);**

The environment itself can make information available to a program. This information often is available in the form of an *environment variable*, which is a string that forms a definition. For example, under the UNIX operating system the environment variable **TERM** indicates the type of terminal the user has. The variable **TERM=myterm** indicates that the user is typing on a *myterm* variety of terminal. When a program reads that declaration, it knows to use the coding proper for that terminal.

The environment variables together form the *environment list*. Given the heterogeneous environments under which C is implemented, the Standard does not define the mechanism by which the environment list is passed to a program.

The function **getenv** scans the environment list and looks for the variable that is named in the string pointed to by *variable*.

getenv returns a pointer to the string that defines the variable. It returns NULL if the variable requested cannot be found.

Example

This program looks up words in the environment and displays them.

```
#include <stdio.h>
#include <stdlib.h>
```

```
main(void)
{
      for(;;) {
            char buf[80], *is;

            printf("Enter an environmental variable: ");
            fflush(stdout);

            if(gets(buf) == NULL)
                  exit(EXIT_SUCCESS);

            if((is = getenv(buf)) == NULL)
                  printf("Can't find %s\n", buf);
            else
                  printf("%s = %s\n", buf, is);
      }

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.4.4
The C Programming Language, ed. 2, p. 253

See Also

environment list, general utilities

Notes

getenv uses a static area to hold the environment variable requested. This buffer will be overwritten by subsequent calls to **getenv**.

gets() — STDIO (libc)

Read a string from the standard input stream
#include <stdio.h>
char ***gets(char** **buffer**);*

gets reads characters from the standard input stream and stores them in the area pointed to by *buffer*. It stops reading as soon as it detects a newline character or the end of file. **gets** discards the newline or **EOF** and appends a null character onto the end of the string it has built.

If all goes well, **gets** returns *buffer*. When it has encountered the end of file without having placed any characters into *buffer*, it returns NULL and leaves the contents of *buffer* unchanged. If a read error occurs, **gets** returns NULL and the contents of *buffer* may or may not be altered.

Example

This example echoes whatever is typed upon the standard-input device.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      char buf[100];

      while(gets(buf) != NULL)
            puts(buf);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.7
The C Programming Language, ed. 2, p. 247

See Also

fgets, getc, getchar, putc, putchar, puts, STDIO, ungetc

Notes

gets stops reading the input string as soon as it detects a newline character. If a previous read from the standard input stream left a newline character in the standard input buffer, **gets** will read it and immediately stop accepting characters. To the user, it will appear as if **gets** is not working at all.

For example, if **getchar** is followed by **gets**, the first character **gets** will receive is the newline character left behind by **getchar**. A simple statement will remedy this:

```
while (getchar() != '\n')
      ;
```

This discards the newline character left behind by **getchar**. **gets** will now work correctly. You should use this only when you know that a newline will be left in the buffer. Otherwise, the desired line will be lost

getw() — Extended function (libc)

Read word from file stream
#include <xstdio.h>
int getw(FILE *_fp_**);**

getw reads a word (an **int**) from the file stream *fp*, and returns it. It differs from the related function **getc** in that **getc** returns either a **char** promoted to an **int**, or EOF.

getw returns EOF on errors; however, you must call **feof** or **ferror** distinguish this value from a valid end-of-file signal.

Example

For an example of this function, see the entry for **inb**.

See Also

extended STDIO, getc

Notes

getw assumes that the bytes of the word it receives are in the natural byte ordering of the machine. See the entry on **byte ordering** for more information. This means that such files might not be portable between machines.

To conform to the ANSI Standard, this function has been moved from the header **stdio.h** to the header **xstdio.h**. This may require that some code be altered.

getw is not described in the ANSI Standard. A program that uses it does not comply strictly with the Standard, and may not be portable to other compilers or operating systems.

gmtime() — Time function (libc)

Convert calendar time to universal coordinated time
#include <time.h>
struct tm ***gmtime(const time_t** ****caltime**);**

The function **gmtime** takes the calendar time pointed to by *caltime* and breaks it down into a structure of the type **tm**, converting it into universal coordinated time.

LEXICON

gmtime returns a pointer to the structure **tm** that it creates. This structure is defined in the header **time.h**. If universal coordinated time cannot be computed, then **gmtime** returns NULL.

Example

This example shows Universal Coordinated Time in a message of the form "12/22/88 15:27:33".

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void)
{
      time_t now;
      char buffer[80];

      time(&now);
      strftime(buffer, sizeof(buffer),
            "%m/%d/%y %H:%M:%S\n", gmtime(&now));
      printf(buffer);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.3.3
The C Programming Language, ed. 2, p. 256

See Also

asctime, ctime, date and time, localtime, strftime, tm, universal coordinated time

Notes

The name "gmtime" reflects the term "Greenwich Mean Time." the Standard prefers the term "universal coordinated time," although for all practical purposes the two are identical.

gmtime is useful only on a system whose time is set to UTC rather than to local time. The **Let's C** time routines read the environmental variable **TIMEZONE** to translate UTC automatically into your local time, should you wish. See the entry for **TIMEZONE** for more information on how this works.

gmtime returns a pointer to a statically allocated data area that is overwritten by successive calls.

goto — C keyword

Unconditionally jump within a function
goto *label***;**

The **goto** statement forces a program's execution to jump to the point marked by *label*. A **goto** can jump only to a point within the current function. To jump beyond a function boundary, use the functions **longjmp** and **setjmp**.

The most common use for **goto** is to exit from nested control structures or go to the top of a control block. It is used most often to write "ripcord" routines, i.e., routines that are executed when a error occurs too deeply within a program for the program to disentangle itself correctly.

Example

For an example of this statement, see **name space**.

Cross-references

Standard, §4.6.6.1
The C Programming Language, ed. 2, p. 65

See Also

break, C keywords, continue, label name, non-local jumps, return, statements

Notes

The C Programming Language describes **goto** as "infinitely-abusable." *Caveat utilitor*.

H

header — Overview

The Standard mandates that every function be declared in a *header*, whose contents are available to the program through the **#include** preprocessor directive. A header usually is a file, but it may also be built into the translator.

The Standard describes 15 headers, as follows:

assert.h	Run-time assertion checking
ctype.h	Character-handling functions
errno.h	**errno** and related macros
float.h	Limits to floating-point numbers
limits.h	General implementation limits
locale.h	Establish or modify a locale
math.h	Mathematics function
setjmp.h	Non-local jumps
signal.h	Signal-handling functions
stdarg.h	Handle variable numbers of arguments
stddef.h	Common definitions
stdio.h	Standard input and output
stdlib.h	General utilities
string.h	String-handling functions
time.h	Date and time functions

Each header contains only those functions described within the Standard, plus attending data types and macros. Every external identifier in every header is reserved for the implementation. Also reserved is every external identifier that begins with an underscore character '_', whether it is described in the Standard or not. If a reserved external name is redefined, behavior is undefined, even if the function that replaces it has the same specification as the original. This is done to assure the user that moving code from one implementation to another will not generate unforeseen collisions with implementation-defined identifiers. It is also done to assure the implementor that functions called by other library functions will not be derailed by user-defined external names.

Every header can be included any number of times, and any number of headers can be included in any order without triggering problems.

Let's C also includes the following, implementation-specific headers:

access.h	Define manifest constants used by **access()**
bios.h	Outline ROM BIOS data area
canon.h	Canonical conversion for the 68000
dos.h	Define MS-DOS functions and devices
larges.h	Support model-independent assembly language
mtype.h	List processor code numbers
path.h	Declare **path()**
stat.h	Definitions and declarations to obtain file status
xctype.h	Declare/define extended character handling routines
xmath.h	Declare extended mathematics functions
xstdio.h	Declare/define extended STDIO routines
xtime.h	Declare/define extended date and time routines

Cross-references

Standard, §4.1.3
The C Programming Language, ed. 2, p. 241

See Also

header names, Library

header names — Definition

A *header name* is a token that gives the name of a header. There are two varieties of header name: **<***filename***.h>** and "*filename***.h"**.

The two varieties of header names are both searched in an implementation-defined manner. The name of the file can be enclosed within angle brackets (**<***file***.h>**) or quotation marks ("*file***.h"**). Angle brackets tell **Let's C** to look for *file***.h** in the directories named with the **-I** options to the **cc** command, and then in the directory named by the environmental variable **INCDIR**. Quotation marks tell **Let's C** to look for *file***.h** in the source file's directory, then in directories named with the **-I** options, and then in the directory named by the environmental variable **INCDIR**.

If any of the characters **'**, ****, **,** or **/*** appear between the '<' and '>' of a bracketed header name, behavior is undefined. Likewise, if any of the characters **'**, ****, or **/*** appear between the "" and the '"' of a quoted header name, behavior is undefined.

Cross-references

Standard, §3.1.7

See Also

#include, header, lexical elements

hypot() — Extended function (libm)

Compute hypotenuse of right triangle
#include <xmath.h>
double hypot(double *x***, double** *y***);**

hypot computes the hypotenuse, or distance from the origin, of its arguments *x* and *y*. The result is the square root of the sum of the squares of *x* and *y*.

See Also

cabs, extended mathematics

Notes

hypot is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

I

i8086 support — Overview

Let's C includes a number of routines that support the i8086 microprocessor. They are as follows:

_copy	Copy memory from one address to another
csreg	Read the CS segment register
dsreg	Read the DS segment register
esreg	Read the ES segment register
exargs	Parse the command line
execall	Pass a command to **command.com**
getanb	Get unbuffered input from **aux** device
getcnb	Get unbuffered input from **con** device
in	Read a word from a port
inb	Read a byte from a port
intcall	Call an MS-DOS interrupt
out	Output a word to a port
outb	Output a byte to a port
ptoreg	Convert C ponters to register pairs
PTR	Expand pointers to offset/segment
putanb	Send unbuffered output to **aux** device
putcnb	Send unbuffered output to **con** device
regtop	Set a pointer to value of register pair
ssreg	Read the SS segment register
_zero	Zero out a segment of memory

See Also

i8087, Library

Notes

These functions are not described in the ANSI Standard. A program that uses any of them does not conform strictly to the Standard, and may not be portable to other compilers or environments.

i8087 — Technical information

Floating-point co-processor

The Intel i8087 is the mathematics coprocessor for the i8086/88 family of microprocessors. It greatly accelerates the computation of floating-point numbers.

Let's C includes two sets of libraries for use with the i8087: the *sensing* and the *non-sensing* libraries.

If your compiled program is always going to run on a computer system that includes an i8087, you should compile and link programs with the **-VNDP** option. This program will use the non-sensing libraries. These libraries contain instructions that perform floating point operations directly on the i8087 coprocessor; programs compiled with them will *not* operate correctly on a system which does not include an i8087.

You should *not* compile and link programs with the **-VNDP** option if your system does not include an i8087 coprocessor or if you want the compiled program to run on target systems that might or might not contain an i8087.

If you do *not* use the **-VNDP** option, **Let's C** by default will use its *sensing* libraries. These libraries check if an i8087 is present on the system on which the compiled program is being run. If one is present, the libraries use it to perform floating-point operations. If one is not present, the libraries emulate i8087 floating-point operations in software. The compiled program will be somewhat larger

than the same program compiled with the **-VNDP** option, because it will include the code to perform software floating point, and will run slightly slower. The program must be linked with the **-VROM** option if it is to run in ROM.

A program that uses floating-point numbers will not necessarily yield the same results when executed on systems with and without an i8087 coprocessor. In particular, the i8087 represents floating-point numbers internally with an 80-bit representation (64 fraction bits, 15 exponent bits, one sign bit), whereas **Let's C** software floating point uses the 64-bit IEEE representation internally (52 fraction bits, 11 exponent bits, one sign bit). Thus, the low-order digits of floating point computations may differ on systems with and without an i8087.

Compatability With Previous Versions

Versions of **Let's C** prior to 4.0 used DECVAX format rather than IEEE format for software floating point operations. Any program using software floating point that was compiled by a previous version must be recompiled with the current version to use IEEE software floating point. Binary files that include DECVAX format floating point data are not compatible with the current IEEE floating point version.

Checking Presence of the i8087

In i8087 sensing mode, the C runtime startup routine discovers whether an i8087 is present on the machine. This datum is written into the global **char _has8087**. Zero indicates that an i8087 is absent, and a value other than zero indicates that it is present.

If you wish, you can read and change this variable. If you wish to test how a program would work without an i8087, it is easier to clear this byte than to pull the i8087 chip out of your computer. If, however, you set this byte to a non-zero value and an i8087 is not present, your computer will hang when it tries to use the non-existent i8087.

See Also

float, double, technical information

Notes

The assembler **as** will assemble programs that use i8087 opcodes. For a full table of these opcodes, see the entry for **as**.

identifiers — Overview

An *identifier* names one of the following lexical elements:

- Functions
- Labels
- Macros
- Members of a structure, a **union**, or an enumeration
- Objects
- Tags
- **typedef**s

An identifier with internal linkage may have up to at least 31 characters, which may be in either upper or lower case. An identifier with external linkage, however, may have up to at least six characters, and it is not required to recognize both upper and lower case. These limits are defined by the implementation, and may be increased by it.

An identifier is a string of digits and non-digits, beginning with a non-digit. For a translator to know that two identifiers refer to the same entity, the identifiers must be identical. If two identifiers are

meant to refer to the same entity yet differ in any character, the behavior is undefined.

Keywords in C are reserved. Therefore, no identifier may match a keyword.

The Standard allows the programmer to use leading underscores '_' to name internal identifiers, but reserves for the implementation all external identifiers with leading underscores. To reduce "name space pollution," the implementor should not reserve anything that is not explicitly defined in the Standard and that does not begin with a leading underscore.

Identifiers have both *scope* and *linkage*. The scope of an identifier refers to the portion of a program to which it is "visible." An identifier can have program scope, file scope, function scope, or block scope; for more information, see the entry for **scope**. The linkage of an identifier describes whether it is joined only with its name-sakes within the same file, or can be joined to other files. Linkage can be external, internal, or none. For more information, see the entry for **linkage**.

Cross-references

Standard, §3.1.2
The C Programming Language, ed. 2, p. 192

See Also

digit, external name, function prototype, internal name, lexical elements, linkage, name space, nondigit, scope, storage duration, string literal, types

if — C keyword

Conditionally execute an expression
if(*conditional***)** *statement***;**

if is a C keyword that conditionally executes an expression. If *conditional* is nonzero, then *statement* is executed. However, if *conditional* is zero, then *statement* is not executed.

conditional must use a scalar type. It may be a function call (in which case **if** evaluates what function returns), an integer, the result of an arithmetic operation, or the value returned by a relational expression.

An **if** statement can be followed by an **else** statement, which also introduces a statement. If *conditional* is nonzero, then the statement introduced by **if** is executed and the one introduced by **else** is ignored; whereas if *conditional* is equal to zero, then the statement introduced by **if** is ignored and the one introduced by **else** is executed.

Example

For an example of this statement, see **exit**.

Cross-references

Standard, §4.6.4.1
The C Programming Language, ed. 2, pp. 55*ff*

See Also

else, statements, switch

Notes

If the statement controlled by an **if** statement is accessed via a label, the statement controlled by an **else** statement associated with the **if** statement is not executed.

implicit conversions — Definition

The term *implicit conversion* means that the type of an object is changed by the translator without the direct intervention of the programmer. For a list of the rules for implicit conversion, see **conversion**.

Cross-reference

Standard, §3.2

See Also

conversions, explicit conversion

inb() — Extended function (libc)

Read from a port
int inb(int *port***);**

inb provides a C interface to the i8086 machine instruction **in**. It reads a byte (eight bits) from *port*, and returns it as an integer (16 bits).

Example

This example writes a file to the serial port. It uses **inb** to read the current status of the port.

```
#include <stdio.h>
#include <stdlib.h>

/* DOS magic numbers */
#define PRINTER_STATUS 0x3BD
#define PRINTER_OUT 0x3BC
#define PRINTER_BUSY 0x80

main(int argc, char *argv[])
{
      FILE *fp;
      int  data;

      if(argc != 2)
            printf("Usage: print filename\n");
      else if ((fp = fopen(argv[1], "r")) == NULL) {
            printf("Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      else while((data = getw(fp)) != EOF) {
            while(inb(PRINTER_STATUS) & PRINTER_BUSY)
                  ;
            outb(PRINTER_OUT, data);
      }
      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, in, out, outb

INCDIR — Environmental variable

Directory that holds include files

INCDIR names the default directory where **Let's C** seeks its header files. For example, the command

LEXICON

```
set INCDIR=a:\include
```

tells **cc** to look for header files in directory **include** on drive A. This directory is searched, as is the directory that holds the C source files and the directories named with **-I** options to the **cc** command, if any.

It is recommended that you set **INCDIR** in **autoexec.bat** to ensure that it is always set correctly.

See Also

cc, environmental variable

index() — Extended function (libc)

Find a character in a string
char *index(char **string***, **char** *character***);**

index is identical to the ANSI function **strchr**. It scans the given *string* for the first occurrence of *character*. If it finds *character*, it returns a pointer to it. If it does not find *character*, **index** returns NULL.

Having **index** search for a null character will always produce a pointer to the end of a string. For example,

```
char *string;
assert(index(string, 0)==string+strlen(string));
```

will never fail.

Example

For an example of this function, see the entry for **strncpy**.

See Also

extended miscellaneous, memchr, pnmatch, rindex, strchr, strpbrk

Notes

index is not described in the ANSI Standard. It is recommended that you use **strchr** instead of **index** so your programs will more closely approach strict conformity with the Standard.

initialization — Definition

The term *initialization* refers to setting a variable to its first, or initial, value.

Rules of Initialization

Initializers follow the same rules for type and conversion as do assignment statements.

If a static object with a scalar type is not explicitly initialized, it is initialized to zero by default. Likewise, if a static pointer is not explicitly initialized, it is initialized to NULL by default. If an object with automatic storage duration is not explicitly initialized, its contents are indeterminate.

Initializers on static objects must be constant expressions; greater flexibility is allowed for initializers of automatic variables. These latter initializers can be arbitrary expressions, not just constant expressions. For example,

```
double dsin = sin(30);
```

is a valid initializer, where **dsin** is declared inside a function.

To initialize an object, use the assignment operator '='. The following sections describe how to initialize different classes of objects.

Scalars

To initialize a scalar object, assign it the value of a expression. The expression may be enclosed within braces; doing so does not affect the value of the assignment. For example, the expressions

```
int example = 7+12;
```

and

```
int example = { 7+12 };
```

are equivalent.

Unions and Structures

The initialization of a **union** by definition fills only its *first* member.

To initialize a **union**, use an expression that is enclosed within braces:

```
union example_u {
      int member1;
      long member2;
      float member3;
} = { 5 };
```

This initializes **member1** to five. That is to say, the **union** is filled with an **int**-sized object whose value is five.

To initialize a structure, use a list of constants or expressions that are enclosed within braces. For example:

```
struct example_s {
      int member1;
      long member2;
      union example_u member3;
};

struct example_s test1 = { 5, 3, 15 };
```

This initializes **member1** to five, initializes **member2** to three, and initializes the *first* member of **member3** to 15.

Strings and Wide Characters

To initialize a string pointer or an array of wide characters, use a string literal.

The following initializes a string:

```
char string[] = "This is a string";
```

The length of the character array is 17 characters: one for every character in the given string literal plus one for the null character that marks the end of the string.

If you wish, you can fix the length of a character array. In this case, the null character is appended to the end of the string only if there is room in the array. For example, the following

```
char string[16] = "This is a string";
```

writes the text into the array **string**, but does not include the concluding null character because there is not enough room for it.

The same rules apply to initializing an array of wide characters. For example, the following:

```
wchar_t widestring[] = L"This is a string";
```

LEXICON

fills **widestring** with the wide characters corresponding to the characters in the given string literal. The appropriate form of the null character is then appended to the end of the array, and the size of the array is **(17*sizeof(wchar_t))**. The prefix **L** indicates that the string literal consists of wide characters.

A pointer to **char** can also be initialized when the pointer is declared. For example:

```
char *strptr = "This is a string";
```

initializes **strptr** to point to the first character in **This is a string**. This declaration automatically allocates exactly enough storage to hold the given string literal, plus the terminating null character.

Arrays

To initialize an array, use a list of expressions that is enclosed within braces. For example, the expression

```
int array[] = { 1, 2, 3 };
```

initializes **array**. Because **array** does not have a declared number of elements, the initialization fixes its number of elements at three. The elements of the array are initialized in the order in which the elements of the initialization list appear. For example, **array[0]** is initialized to one, **array[1]** to two, and **array[2]** to three.

If an array has a fixed length and the initialization list does not contain enough initializers to initialize every element, then the remaining elements are initialized in the default manner: static variables are initialized to zero, and other variables to whatever happens to be in memory. For example, the following:

```
int array[3] = { 1, 2 };
```

initializes **array[0]** to one, **array[1]** to two, and **array[2]** to zero.

The initialization of a multi-dimensional array is something of a science in itself. The Standard defines that the ranks in an array are filled from right to left. For example, consider the array:

```
int example[2][3][4];
```

This array contains two groups of three elements, each of which consists of four elements. Initialization of this array will proceed from **example[0][0][0]** through **example[0][0][3]**; then from **example[0][1][0]** through **example[0][1][3]**; and so on, until the array is filled.

It is easy to check initialization when there is one initializer for each "slot" in the array; e.g.,

```
int example[2][3] = {
       1,  2,  3,  4,  5,  6
};
```

or:

```
int example[2][3] = {
      {  1,  2,  3 }, {  4,  5,  6 }
};
```

The situation becomes more difficult when an array is only partially initialized; e.g.,

```
int example[2][3] = {
       { 1 }, { 2, 3 }
};
```

which is equivalent to:

```
int example[2][3] = {
      { 1, 0, 0 }, { 2, 3, 0 }
};
```

As can be seen, braces mark the end of initialization for a "cluster" of elements within an array. For example, the following:

```
int example[2][3][4] = {
      5, { 1, 2 }, { 5, 2, 4, 3 }, { 9, 9, 5 },
      { 2, 3, 7 } };
```

is equivalent to entering:

```
int example[2][3][4] = {
      { 5, 0, 0, 0 },
      { 1, 2, 0, 0 },
      { 5, 2, 4, 3 },

      { 9, 9, 5, 0 },
      { 2, 3, 7, 0 },
      { 0, 0, 0, 0 }
};
```

The braces end the initialization of one cluster of elements; the next cluster is then initialized. Any elements within a cluster that have not yet been initialized when the brace is read are initialized in the default manner.

The final entry in a list of initializers may end with a comma. For example:

```
int array[3] = { 1, 2, 3, };
```

will initialize **array** correctly. This is a departure from many current implementations of C.

ANSI C requires that the initializers of a multi-dimensional array be parsed in a top-down manner. Some implementations had parsed such initializers in a bottom-up manner. Code that expects bottom-up parsing may behave differently under ANSI C, and probably without warning. This is a quiet change that may require that some code be rewritten.

Cross-references

Standard, §3.5.7
The C Programming Language, ed. 2, pp. 218*ff*

See Also

array, declarations

int — C keyword

The type **int** holds an integer. It is usually the same size as a word (or register) on the target machine.

int is a signed integral type. This type can be no smaller than an **short** and no greater than a **long**.

A **int** can encode any number between **INT_MIN** and **INT_MAX**. These are macros that are defined in the header **limits.h**;

The types **signed** and **signed int** are synonyms for **int**.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 211

LEXICON

See Also

types

Notes

Because **int**s may be the size of **short**s on some machines and the size of **long**s on others, programs that are meant to be portable can avoid bugs by explicitly declaring all **int**s to be either **short** or **long**.

intcall() — i8086 support (libc)

Call MS-DOS interrupt
#include <dos.h>
int intcall(struct reg **srcreg***, struct reg** **destreg***, int** *intnum***);**

intcall lets you call MS-DOS interrupts. The arguments *srcreg* and *destreg* point to elements in the structure **reg**, which is defined in the header file **dos.h**, as follows:

```
struct reg {
      unsigned r_ax;
      unsigned r_bx;
      unsigned r_cx;
      unsigned r_dx;
      unsigned r_si;
      unsigned r_di;
      unsigned r_ds;
      unsigned r_es;
      unsigned r_flags;
};
```

intcall sets the processor registers to the values given in *srcreg*, without setting the processor flags. Then it calls the interrupt specified by *intnum* to perform the desired system function. Most often, the manifest constant **DOSINT** (0x21) is used, although **intcall** can handle almost all MS-DOS interrupts. Finally, it sets the structure pointed to by *destreg* to the values of those registers, and returns.

Example

The following program uses function 8 of interrupt 21, which receives raw input from the keyboard and does not echo it on the screen. The program receives up to 80 characters typed at the keyboard, and echoes them to the screen either when the carriage return is pressed or when the limit of 80 characters is exceeded.

The sample program **fdir.c**, which is included with your copy of **Let's C**, also demonstrates **intcall**.

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

char getch(void)
{
      struct reg r;

      r.r_ax = CONRAW;
      intcall(&r, &r, DOSINT);

      /* mask off top of ax pair */
      return (r.r_ax & 0xff);
}
```

```
main(void)
{
      char string[80];
      int i;

      for(i = 0; i <80; i++)
            if((string[i] = getch()) == '\r')
                  break;

      printf("%s\n", string);
      return(EXIT_SUCCESS);
}
```

See Also

dos.h, i8086 support, ptoreg, PTR, regtop, signals/interrupts

Notes

Registers that are not included in the structure **reg** cannot be passed to a system routine explicitly.

incall cannot use interrupts 25 and 26, absolute disk read and write, because they do not restore the stack correctly when they exit.

integer constant — Definition

An *integer constant* is a constant that holds an integer. An integer constant has the following structure:

- It begins with a digit.
- It has no period or exponent.
- It may have a prefix that indicates its base, as follows: **0X** and **0x** both indicate hexadecimal. **0** (zero) indicates octal.
- It may have a suffix that indicates its type. **u** and **U** indicate an unsigned integer; **l** and **L** indicate a long integer.

A hexadecimal number may consist of the digits '0' through '9' and the letters 'a' through 'f' or 'A' through 'F'. An octal number may consist of the digits '0' through '7'.

When an integer constant initializes a variable, the form of the constant should match that of the variable as closely as possible. For example, when an integer constant initializes a **long int**, the constant should have the suffix **l** or **L**. If the constant does not have this suffix, the variable may not be initialized correctly.

The type of an integer constant is fixed by the following rules:

- A decimal integer constant that has no suffix is given the *first* of the following types that can represent its value: **int**, **long int**, or **unsigned long int**.
- A hexadecimal or octal integer constant that has no suffix is given the first of the following types that can represent its value: **int**, **unsigned int**, **long int**, or **unsigned long int**.
- An integer constant with the prefixes **u** or **U** is given the first of the following types that can represent its value: **unsigned int** or **unsigned long int**.
- An integer constant with the prefixes **l** or **L** is given the first of the following types that can represent its value: **long int** or **unsigned long int**.
- An integer constant with both the unsigned and the long suffixes is an **unsigned long int**.

These rules, as they preserve the value of a given constant, are part of what is known as the *value-*

preserving rules.

Cross-references

Standard, §3.1.3.2
The C Programming Language, ed. 2, p. 193

See Also

constants, conversions

internal name — Definition

An *internal name* is an identifier that has internal linkage. The minimum maximum for the length of an internal name is 31 characters, and an implementation must distinguish upper-case and lower-case characters.

Cross-references

Standard, §3.1.2
The C Programming Language, ed. 2, p. 35

See Also

external name, identifiers, linkage

interrupt — Definition

An *interrupt* is an interruption of the sequential flow of a program. It can be generated by the hardware, from within the program itself, or from the operating system.

See Also

Definitions, intcall, interrupt handling, interrupts

isalnum() — Character handling (ctype.h)

Check if a character is a numeral or letter
#include <ctype.h>
int isalnum(int *c***);**

The macro **isalnum** tests whether *c* is a letter or a numeral. A letter is any character for which **isalpha** returns true; likewise, a numeral is any character for which **isdigit** returns true. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isalnum returns nonzero if *c* is a letter or a numeral, and zero if it is not.

Cross-references

Standard, §4.3.1.1
The C Programming Language, ed. 2, pp

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isalpha() — Character handling (ctype.h)

Check if a character is a letter
#include <ctype.h>
int isalpha(int *c***);**

The macro **isalpha** tests whether *c* is a letter. In the **C** locale, a letter is any of the characters 'a' through 'z' or 'A' through 'Z'. In any other locale, a letter is any character for which the functions **iscntrl**, **isdigit**, **ispunct**, and **isspace** all return false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isalpha returns nonzero if *c* is an alphabetic character, and zero if it is not.

Cross-references

Standard, §4.3.1.2
The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isascii() — Extended macro (xctype.h)

Check if a character is an ASCII character
#include <xctype.h>
int isascii(*c***) int** *c***;**

The macro **isascii** tests whether the argument *c* is an ASCII character (0 <= *c* <= 0177). It returns a number other than zero if *c* is an ASCII character, and zero if it is not. Many other **ctype** macros will fail if passed a non-ASCII value other than **EOF**.

See Also

extended character handling

Notes

To conform to the ANSI Standard, this macro has been moved from the header **ctype.h** to the header **xctype.h**. This may require that some code be altered.

This macro is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

iscntrl() — Character handling (ctype.h)

Check if a character is a control character
#include <ctype.h>
int iscntrl(int *c***);**

The macro **iscntrl** tests whether *c* is a control character under the implementation's character set. The Standard defines a control character as being a character in the implementation's character that cannot be printed. *c* must be a value that is representable as an **unsigned char** or **EOF**.

iscntrl returns nonzero if *c* is a control character, and zero if it is not.

Cross-references

Standard, §4.3.1.3

LEXICON

The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isdigit() — Character handling (ctype.h)

Check if a character is a numeral
#include <ctype.h>
int isdigit(int *c***);**

The macro **isdigit** tests whether *c* is a numeral (any of the characters '0' through '9'). *c* must be a value that is representable as an **unsigned char** or **EOF**.

isdigit returns nonzero if *c* is a numeral, and zero if it is not.

Cross-references

Standard, §4.3.1.4
The C Programming Language, ed. 2, p. 249

See Also

character handling

isgraph() — Character handling (ctype.h)

Check if a character is printable
#include <ctype.h>
int isgraph(int *c***);**

The macro **isgraph** tests whether *c* is a printable letter within the **Let's C** character set, but excluding the space character. The Standard defines a printable character as any character that occupies one printing position on an output device. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isgraph returns nonzero if *c* is a printable character (except for space), and zero if it is not.

Cross-references

Standard, §4.3.1.5
The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

islower() — Character handling (ctype.h)

Check if a character is a lower-case letter
#include <ctype.h>
int islower(int *c***);**

The macro **islower** tests whether *c* is a lower-case letter. In the **C** locale, a lower-case letter is any of the characters 'a' through 'z'. In any other locale, this is a character for which the functions **iscntrl**,

isdigit, **ispunct**, **isspace**, and **isupper** all return false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

islower returns nonzero if *c* is is a lower-case letter, and zero if it is not.

Cross-references

Standard, §4..1.6
The C Programming Language, ed. 2, p. 249

See Also

character handling, character set

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isprint() — Character handling (ctype.h)

Check if a character is printable
#include <ctype.h>
int isprint(int *c***);**

The macro **isprint** tests whether *c* is a printable letter within the implementation's character set, including the space character. The Standard defines a printable character as any character that occupies one printing position on an output device. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isprint returns nonzero if *c* is a printable character, and zero if it is not.

Cross-references

Standard, §4.3.1.7
The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

ispunct() — Character handling (ctype.h)

Check if a character is a punctuation mark
#include <ctype.h>
int ispunct(int *c***);**

The macro **ispunct** tests whether *c* is a punctuation mark in the implementation's character set. The Standard defines a punctuation mark as being any printable character, except the space character, for which the function **isalnum** returns false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

ispunct returns nonzero if *c* is a punctuation mark, and zero if it is not.

Cross-references

Standard, §4.3.1.8
The C Programming Language, ed. 2, p. 249

LEXICON

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isspace() — Character handling (ctype.h)

Check if character is white space
#include <ctype.h>
int isspace(int *c***);**

The macro **isspace** tests whether *c* represents a white-space character. In the **C** locale, a white-space character is any of the following: space (' '), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v'). In any other locale, a white-space character is one for which the functions **isalnum**, **iscntrl**, **isgraph**, and **ispunct** all return false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isspace returns nonzero if *c* is a space character, and zero if it is not.

Cross-references

Standard, §4.3.1.1
The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. For example, Middle-Eastern languages use alternate characters to denote white space. See **localization** for more information.

isupper() — Character handling (ctype.h)

Check if a character is an upper-case letter
#include <ctype.h>
int isupper(int *c***);**

The macro **isupper** tests whether *c* is a upper-case letter. In the **C** locale, a upper-case letter is any of the characters 'A' through 'Z'. In any other locale, this is a character for which the functions **iscntrl**, **isdigit**, **islower**, **ispunct**, and **isspace** all return false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isupper returns nonzero if *c* is an upper-case letter, and zero if it is not.

Cross-references

Standard, §4.3.1.6
The C Programming Language, ed. 2, p. 249

See Also

character handling, character sets

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isxdigit() — Character handling (libc)

Check if a character is a hexadecimal numeral
#include <ctype.h>
int isxdigit(int *c***);**

isxdigit tests whether *c* is a hexadecimal numeral (any of the characters '0' through '9', any of the letters 'a' through 'd', or any of the letters 'A' through 'D'). *c* must be a value that is representable as an **unsigned char** or **EOF**.

isxdigit returns nonzero if *c* is a hexadecimal numeral, and zero if it is not.

Cross-references

Standard, §4.3.1.11
The C Programming Language, ed. 2, p. 249

See Also

character handling

J

j0() — Extended function (libm)

Compute Bessel function
#include <xmath.h>
double j0(double *z***);**

j0 computes the Bessel function of the first kind for order 0, for its argument *z*.

Example

This example, called **bessel.c**, demonstrates the Bessel functions **j0**, **j1**, and **jn**. Compile it with the following command line

```
cc -f bessel.c -lm
```

to include floating-point functions and the mathematics library.

```
#include <math.h>
#include <stdlib.h>
#include <xmath.h>
dodisplay(double value, char *name)
{
      if (errno)
            perror(name);

      else
            printf("%10g %s\n", value, name);
      errno = 0;
}

#define display(x) dodisplay((double)(x), #x)

main()
{
      extern char *gets();
      double x;
      char string[64];

      for(;;) {
            printf("Enter number: ");
            if(gets(string) == 0)
                  break;
            x = atof(string);

            display(x);
            display(j0(x));
            display(j1(x));
            display(jn(0,x));

            display(jn(1,x));
            display(jn(2,x));
            display(jn(3,x));
      }
}
```

See Also

extended mathematics, j1, jn

Notes

j0 is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

j1() — Extended function (libm)

Compute Bessel function
#include <xmath.h>
double j1(double *z***);**

j1 takes the argument *z* and computes the Bessel function of the first kind for order 1.

Example

For an example of this function, see the entry for **j0**.

See Also

extended mathematics, j0, jn

Notes

j1 is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

jday_to_time() — Extended function (libc)

Convert Julian date to system time
#include <time.h>
#include <xtime.h>
time_t jday_to_time(jday_t *time***);**

jday_to_time converts Julian time to system time.

time is the Julian time to be converted. It is of type **jday_t**, which is defined in the header **xtime.h**. **jday_t** is a structure that consists of two **unsigned long**s. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

jday_to_time returns the Julian time as converted to type **time_t**. This type is defined in the header **time.h** as being equivalent to a **long**. **Let's C** defines the current system time as being the number of seconds from January 1, 1970, 0h00m00s GMT, which is equivalent to the Julian day 2,440,587.5.

See Also

extended time, jday_to_tm, time_to_jday, tm_to_jday, xtime.h

Note

This function is of use mainly to astronomers, geographers, and historians.

To conform to the ANSI Standard, this function has been moved from the header **time.h** to the header **xtime.h**. This may require that some code be altered.

jday_to_tm() — Extended function (libc)

Convert Julian date to system calendar format
#include <time.h>
#include <xtime.h>
tm *jday_to_tm(jday_t *time***);**

jday_to_tm converts Julian time to the system calendar format.

time is the Julian time to be converted. It is of type **jday_t**, which is defined in the header **xtime.h**. **jday_t** is a structure that consists of two **unsigned long**s. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.).

The second gives the number of seconds since midnight of the given Julian day.

jday_to_tm returns a pointer to a copy of the structure **tm**, which is defined in the header file **time.h**. For more information on this structure, see the Lexicon entry for **time**.

See Also

extended time, jday_to_time, time_to_jday, tm_to_jday, xtime.h

Note

This function is of use mainly to astronomers, geographers, and historians.

To conform to the ANSI Standard, this function has been moved from the header **time.h** to the header **xtime.h**. This may require that some code be altered.

jmp_buf — Type

Type used with non-local jumps
#include <setjmp.h>

jmp_buf is a type defined in the header **setjmp.h**. It is the type used to hold the current environment to enable a non-local jump. The usual contents of the **jmp_buf** array will be the contents of registers; however, its contents are defined by the implementation.

Cross-references

Standard, §4.6
The C Programming Language, ed. 2, p. 254

See Also

non-local jumps, setjmp.h

Notes

Because **jmp_buf** usually does not contain anything except the current contents of the registers, one should not expect values of local variables or register variables to restored properly.

Historically, code has been written that calls **setjmp** and **longjmp** with an argument of type **jmp_buf**, but without taking its address. This code works because an array passed as a parameter is automatically converted to a pointer. Because structures can now be passed by value, such arguments are no longer converted to pointers. However, because both **setjmp** and **longjmp** expect a pointer argument, the type of **jmp_buf** is restricted to an array type in order to preserve existing code.

If **jmp_buf** must be a structure of heterogeneous elements, then it could be defined as a one-element array of such structures.

jn() — Extended function (libm)

Compute Bessel function
#include <xmath.h>
double jn(short *n*, **double** *z*);

jn takes an argument *z* and computes the Bessel function of the first kind for order *n*.

Example

For an example of this function, see the entry for **j0**.

See Also

extended mathematics, j0, j1

Notes

jn is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

K

keywords — Definition

A *keyword* is a word that has special significance to the C language. All keywords are reserved; none may be used as an identifier.

The Standard defines the following as being C keywords:

auto	**break**	**case**	**char**
const	**continue**	**default**	**defined**
do	**double**	**else**	**enum**
extern	**float**	**for**	**goto**
if	**int**	**long**	**register**
return	**short**	**signed**	**sizeof**
static	**struct**	**switch**	**typedef**
union	**unsigned**	**void**	**volatile**
while			

Let's C also recognizes the keyword **alien**, which indicates that a function uses non-C calling conventions.

Cross-references

Standard, §3.1.1
The C Programming Language, ed. 2, p. 192

See Also

lexical elements

Notes

The keywords **const**, **enum**, **signed**, **void**, and **volatile** are new to the C language, although some or all of these have been used as common extensions to C. A program that uses any of these words as an identifier may not translate properly under an implementation that conforms to the Standard. Likewise, the Standard eliminates the keyword **entry**, which the first edition of *The C Programming Language* defined as being unused.

The Standard recognizes that the keywords **asm** and **fortran** are common extensions to the C language, and are recognized as such by many implementations of C.

L

label — Definition

A *label* is an identifier followed by a colon ':' or that follows a **goto** statement. It marks a point within a function to which a **goto** statement can jump.

Cross-references

Standard, §3.1.2.6
The C Programming Language, ed. 2, pp. 65

See Also

goto, name space

labs() — General utility (libc)

Compute the absolute value of a long integer
#include <stdlib.h>
int labs(long *n***);**

labs computes the absolute value of the long integer *n*. The *absolute value* of a number is its distance from zero. This is *n* if *n***>=0**, and *-n* otherwise.

Cross-references

Standard, §4.10.6.3
The C Programming Language, ed. 2, p. 253

See Also

abs, general utilities

Language — Overview

The description of the language, both in the Standard and in this Lexicon, has the following topics, which describe completely the syntax and semantics of the language:

- constant expressions
- conversions
- declarations
- expressions
- external definitions
- lexical elements
- preprocessing
- statements

Each of these topics is introduced by its own Lexicon article.

Implementation of the C Language

The following summarizes how **Let's C** implements the C language.

Identifiers:

Characters allowed: **A-Z**, **a-z**, _, **0-9**
Case sensitive.
Number of significant characters in a variable name:
at compile time: **128**
at link time: **16**
Appends '_' to end of external identifiers

Reserved identifiers (keywords):

alien	**extern**	**signed**
auto	**float**	**sizeof**
break	**for**	**static**
case	**goto**	**struct**
char	**if**	**switch**
continue	**int**	**typedef**
const	**long**	**union**
default	**readonly**	**unsigned**
do	**register**	**void**
double	**return**	**volatile**
else	**short**	**while**
enum		

In conformity with the proposed ANSI standard, the keyword **entry** is no longer recognized. The keywords **const** and **volatile** are now recognized, but not implemented. The compiler will produce a warning message if the keyword **volatile** is used with the peephole optimizer.

Data formats (in bits):

char	8
double	64
float	32
int	16
long	32
long double	64
pointer (SMALL model)	16
pointer (LARGE model)	32
short	16
unsigned char	8
unsigned int	16
unsigned long	32
unsigned short	16

float *format:*
IEEE floating point format:
1 sign bit
8-bit exponent
24-bit normalized fraction with hidden bit
IEEE double format:
1 sign bit
11-bit exponent
52-bit fraction
Reserved values:
+- infinity, -0
All floating-point operations are done as **double**s.
Note that this will change when the ANSI standard is adopted.

Limits:
Maximum bitfield size: 16 bits
Maximum number of **case**s in a **switch**: no formal limit
Maximum block nesting depth: no formal limit
Maximum parentheses nesting depth: no formal limit
Maximum structure size: 64 kilobytes
Maximum array size: 64 kilobytes

Structure name-spaces:
Supports both Berkeley, and Kernighan and Ritchie conventions for structure in union.

Register variables:
Two available for **int**s (SMALL and LARGE models)
Two available for pointers (SMALL model only)

Function linkage:
Return values for **int**s: AX
Return values for **long**s: DX:AX
Return values for SMALL-model pointers: AX
Return values for LARGE-model pointers: DX:AX
Return values for **double**s in DX:AX
Parameters pushed on stack in reverse order, **char**s and **short**s pushed as words, **long**s and pointers pushed as **long**s, structures copied onto stack
Caller must clear parameters off stack
Stack frame linkage is done through SP register

Register usage:

AX:	returned **int**s and SMALL-model pointers
BP:	Frame pointer
DI:	register variable (**int** or SMALL-model pointer)
DX:AX:	returned **long**s and LARGE-model pointers
SI:	register variable (**int** or SMALL-model pointer)

Note that registers not described above (BX, CX, DX, plus DS and ES in LARGE model) may be freely overwritten by code that the compiler generates. Programs that include assembly-language modules should take this into account.

Special features and optimizations:

LEXICON

- Branch optimization is performed: this uses the smallest branch instruction for the required range.
- Unreached code is eliminated.
- The contents of word registers are remembered by a peephole optimizer, to avoid reloading.
- Duplicate instruction sequences are removed.
- Jumps to jumps are eliminated.
- Multiplication and division by constant powers of two are changed to shifts when the results are the same.
- Sequences that can be resolved at compile time are identified and resolved.

Cross-references

Standard, §3.0
The C Programming Language, ed. 2, pp. 191*ff*

See Also

byte ordering, declarations, function calls, keywords, Lexicon, Library, memory allocation, types

LARGE model — Technical information

Intel multi-segment memory model

The i8086/88 microprocessor uses a *segmented architecture*. This means that memory is divided into segments of 64 kilobytes each. No program or data element can exceed that limit.

Intel Corporation has devised a number of memory models for handling segmented memory. **Let's C** implements the two most useful of these: SMALL model and LARGE model.

In LARGE model, pointers consist of an offset and a segment. The address is calculated by left-shifting the segment by four and adding the offset. Thus, LARGE model programs can access up to 1,048,576 bytes (one megabyte) of code and data. Because of the design of the IBM PC and compatibles, however, the practical limit of memory is 640 kilobytes.

In terms of execution, LARGE-model programs are less efficient than SMALL-model programs, but for many purposes the advantages of the expanded address space of the LARGE model outweigh the decreased efficiency.

When the **-VLARGE** option is used with the **cc** command, the object program follows the rules of the LARGE model. When you compile a program with the **-VLARGE** option, **cc** defines the global variable **LARGE** to the C preprocessor. This allows you to use the preprocessor statement **#ifdef LARGE** to flag model-dependent code.

See Also

model, pointer, SMALL model, technical information

LC_ALL — Manifest constant

All locale information
#include <locale.h>

LC_ALL is a manifest constant that is defined in the header **locale.h**. When passed to the function **setlocale**, it queries or sets all information for a given locale. Information obtained with this macro alters the operation of all functions that are affected by the program's locale, as well as the contents of the structure **lconv**. The following lists the functions affected by **LC_ALL**:

Collation
strcoll
strxfrm

ctype
isdigit
isxdigit

Date and time
strftime

Formatted I/O
fprintf
fscanf
printf
sprintf
scanf
sscanf
vfprintf
vprintf
vsprintf

Multibyte characters
mblen
mbstowcs
mbtowc
wcstombs
wctomb

String conversion
atof
atoi
atol
strtod
strtol
strtoul

Cross-reference

Standard, §4.4

See Also

LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, lconv, localization, locale.h, setlocale

LC_COLLATE — Manifest constant

Locale collation information
#include <locale.h>

LC_COLLATE is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it queries or sets collation information for a given locale.

This information can affect the operation of the functions **strcoll** and **strxfrm**.

Cross-reference

Standard, §4.4

LEXICON

See Also

LC_ALL, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, localization, locale.h, setlocale

LC_CTYPE — Manifest constant

Locale character-handling information
#include <locale.h>

LC_CTYPE is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it sets or queries the character-handling information for a given locale. This information helps determine the action of the functions declared in **ctype.h**, except **isdigit** and **isxdigit**, as well as the multiple-byte character functions **mblen**, **mbstowcs**, **mbtowc**, **wcstombs**, and **wctomb**.

Cross-reference

Standard, §4.4

See Also

LC_ALL, LC_COLLATE, LC_MONETARY, LC_NUMERIC, LC_TIME, lconv, localization, locale.h, setlocale

LC_MONETARY — Manifest constant

Locale monetary information
#include <locale.h>

LC_MONETARY is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it queries or sets the monetary information for a given locale.

It affects all of the fields within the structure **lconv**, except **decimal_point**.

Cross-reference

Standard, §4.4

See Also

LC_ALL, LC_COLLATE, LC_CTYPE, LC_NUMERIC, LC_TIME, localization, locale.h, setlocale

LC_NUMERIC — Manifest constant

Locale numeric information
#include <locale.h>

LC_NUMERIC is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it queries or sets the information for formatting numeric strings.

This information will alter the operation of the following functions:

Formatted I/O
fprintf
fscanf
printf
sprintf
scanf
sscanf
vfprintf
vprintf
vsprintf

String conversion
atof
atoi
atol
strtod
strtol
strtoul

This information also affects the following fields within the structure **lconv**:

decimal_point
thousands_sep
grouping

Cross-reference

Standard, §4.4

See Also

LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_TIME, lconv, localization, locale.h, setlocale

LC_TIME — Manifest constant

Locale time information
#include <locale.h>

LC_TIME is a manifest constant that is defined in the header **locale.h**. When used with the function **setlocale**, it queries or sets the information for formatting time strings.

This information affects the operation of the function **strftime**.

Cross-reference

Standard, §4.4

See Also

LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, lconv, localization, locale.h, setlocale

lconv — Type

Hold monetary conversion information
#include <locale.h>

lconv is a structure that is defined in the header **locale.h**. Its members hold many details needed to format monetary and non-monetary numeric information for a given locale.

To initialize **lconv** for any given locale, use the function **localeconv**. To change any aspect of the locale information being used, use the function **setlocale.**

lconv contains the following fields:

char *currency_symbol
This points to a string that contains the symbol used locally to represent currency, e.g., the '$'. The **C** locale sets this to point to a null string.

char *decimal_point
This points to a string that contains the character used to indicate the decimal point. The **C** locale sets this to point to '.'.

LEXICON

char frac_digits
This is the number of fractional digits that can be displayed in a monetary string. The **C** locale sets this to **CHAR_MAX**.

char grouping
This points to the string that indicates the grouping characteristics for non-monetary amounts. Characters in the string can take the following values:

0	Use previous element for rest of digits
MAX_CHAR	Perform no further grouping
2 through **9**	No. of digits in current group

The **C** locale sets this to **CHAR_MAX**.

char *int_curr_symbol
This points to a string that contains the international currency symbol for the locale, as defined in the publication *ISO 4217 Codes for Representation of Currency and Funds*. The **C** locale sets this to point to a null string.

char *mon_decimal_point
This points to a string that contains the character used to indicate a decimal point in monetary strings. The **C** locale sets this to point to a null string.

char mon_grouping
This points to the string of characters that indicate the grouping characteristics for monetary amounts. Elements can take the following values:

0	Use previous element for rest of digits
MAX_CHAR	Perform no further grouping
2 through **9**	No. of digits in current group

The **C** locale sets this to **CHAR_MAX**.

char *mon_thousands_sep
This points to a string that contains the character used to separate groups of thousands in monetary strings. The **C** locale sets this to point to a null string.

char n_cs_precedes
This indicates whether the symbol that indicates a negative monetary value precedes or follows the numerals in the monetary string. Zero indicates that it follows the numerals and one indicates that it precedes them. The **C** locale sets this to **CHAR_MAX**.

char n_sep_by_space
This indicates whether a space should appear between the symbol that indicates a negative monetary value and the numerals of the monetary string. Zero indicates that it should not appear, and one indicates that it should. The **C** locale sets this to **CHAR_MAX**.

char n_sign_posn
This indicates the position and formatting of the symbol that indicates a negative monetary value, as follows:

0	Set parentheses around numerals and monetary symbol
1	Set negative sign before currency symbol and numerals
2	Set negative sign after currency symbol and numerals
3	Set negative sign immediately before monetary symbol
4	Set negative sign immediately after monetary symbol

The **C** locale sets this to **CHAR_MAX**.

char *negative_sign
This points to a string that contains the character that indicates a negative value in a monetary string. The **C** locale sets this to point to a null string.

char p_cs_precedes
This indicates whether the currency symbol should precede or follow the numerals in the string. Zero indicates that it precedes the digits and one indicates that it follows. The **C** locale sets this to **CHAR_MAX**.

char p_sep_by_space
This indicates whether a space should appear between the monetary symbol and the numerals of the monetary string. Zero indicates that a space should not appear, and one indicates that it should. The **C** locale sets this to **CHAR_MAX**.

char p_sign_posn
This indicates the position and formatting of the symbol that indicates a positive monetary value, as follows:

0	Set parentheses around numerals and monetary symbol
1	Set positive sign before currency symbol and numerals
2	Set positive sign after currency symbol and numerals
3	Set positive sign immediately before monetary symbol
4	Set positive sign immediately after monetary symbol

The **C** locale sets this to **CHAR_MAX**.

char *positive_sign
This points to a string that contains the character that indicates a non-negative value in a monetary string. The **C** locale sets this to point to a null string.

char *thousands_sep
This points to a string that contains the character used to separate groups of thousands. The **C** locale sets this to point to a null string.

Cross-reference

Standard, §4.4, §4.4.2.1

See Also

CHAR_MAX, locale.h, localeconv, localization, setlocale

ldexp() — Mathematics (libm)

Load floating-point number
#include <math.h>
double ldexp(double *number*, **int** *n*);

ldexp returns *number* times two to the *n* power.

See **float.h** for more information on the structure of a floating-point number.

Cross-references

Standard, §4.5.4.3
The C Programming Language, ed. 2, p. 251

See Also

exp, frexp, log, log10, mathematics, modf

LEXICON

ldiv() — General utility (libc)

Perform long integer division
#include <stdlib.h>
ldiv_t ldiv(long int *numerator*, **long int** *denominator*);

ldiv divides *numerator* by *denominator*. It returns a structure of the type **ldiv_t**, which consists of two **long** members, one named **quot** and the other **rem**. **ldiv** writes the quotient into one **long**, and it writes the remainder into the other.

The sign of the quotient is positive if the signs of the arguments are the same; it is negative if the signs of the arguments differ. The sign of the remainder is the same as the sign of the numerator.

If the remainder is non-zero, the magnitude of the quotient is the largest integer less than the magnitude of the algebraic quotient. This is not guaranteed by the operators **/** and **%**, which merely do what the machine implements for divide.

Example

This example selects one random card out of a pack of 52.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void)
{
      ldiv_t card;

      card = ldiv((unsigned long)time(NULL) % 52, 13L);
      printf("%c%c\n",
            /* note useful addressing for strings */
            "A23456789TJQK"[card.rem],
            "HCDS"[card.quot]);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.6.2
The C Programming Language, ed. 2, p. 253

See Also

/, div, general utilities, ldiv_t

Notes

The Standard includes this function to provide a useful feature of FORTRAN. Also, on most machines, division produces a remainder. This allows a quotient and remainder to be returned from one machine-divide operation.

If the result of division cannot be represented (e.g., because *denominator* is set to zero), the behavior of **ldiv** is undefined.

ldiv_t — Type

Type returned by ldiv()
#include <stdlib.h>

ldiv_t is a typedef that is declared in the header **stdlib.h** and is the type returned by the function **ldiv**.

ldiv_t is a structure that consists of two **long** members, one named **quot** and the other **rem**. **ldiv** writes its quotient into **quot** and its remainder into **rem**.

Example

For an example of this type in a program, see **ldiv**.

Cross-references

Standard, §4.10.6.2
The C Programming Language, ed. 2, p. 253

See Also

general utilities, integer arithmetic, ldiv, stdlib.h

lexical elements — Overview

A *lexical element* is one of the elements from which a C program is built. It is the smallest unit with which a translator can work. "Lexical" refers to the fact that a program is partitioned into tokens during a translation phase that is usually called "lexical analysis."

A C program is built from the following lexical elements:

constants
header names
identifiers
keywords
operators
preprocessing numbers
punctuators
string literals

Cross-reference

Standard, §3.1

See Also

comment, constant, header name, identifier, keyword, Language, operators, preprocessing number, punctuators, string literal, token

Lexicon — Introduction

The Mark Williams Lexicon is a new approach to documentation of computer software. The Lexicon is designed to improve documentation and eliminate some limitations found in more conventional documentation.

How to Use the Lexicon

The Lexicon consists of one large document that contains entries for every aspect of **Let's C**. You will not have to search through a number of different manuals to find the entry you are looking for.

Every entry in the Lexicon has the same structure. The first line gives the name of the topic being discussed, followed by its type (e.g., **Mathematics**) and, where appropriate, the file in which it is kept.

The next lines briefly describe the item, then give the item's usage, where applicable. These are followed by a brief discussion of the item, and an example.

Cross-references follow. These can be to other entries or to other texts, notably to the ANSI Standard, *The Art of Computer Programming* and the second edition of *The C Programming Language*. Diagnostics and notes, where applicable, conclude each entry.

LEXICON

Internally, the Lexicon has a tree structure. The "root" entry is the present entry, for **Lexicon**. Below this entry comes the set of *Overview* entries. Each Overview entry introduces a group of entries; for example, the Overview entry for **string** introduces all of the string functions and macros, lists them, and gives a lengthy example of how to use them.

Each entry cross-references other entries. These cross-references point up the documentation tree, toward an overview article and, ultimately, to the entry for **Lexicon** itself. They also point down the tree to subordinate entries, and across to entries on related subjects. For example, the entry for **getchar** cross-references **STDIO**, which is its Overview article, plus **putchar** and **getc**, which are related entries of interest to the user. The Lexicon is designed so that you can trace from any one entry to any other, simply by following the chain of cross-references up and down the documentation tree.

Use the Lexicon

If, while reading an entry, you encounter a technical term that you do not understand, look it up in the Lexicon. You should find an entry for it. For example, if a function is said to return a data type **float** and you do not know exactly what a **float** is, look it up. You will find it described in full. In this way, you should increase your understanding of **Let's C**, and make your programming easier and more productive.

libcxs87.lib — Library

Standard library, SMALL model/i8087 only

libcxs87.lib is the archive file that holds the SMALL-model version of the more commonly used C functions, system calls, and compiler run-time support routines.

The routines in this library use the i8087 exclusively. They cannot be run on a computer that does not contain an i8087.

To edit this library or create a table of its contents, use the librarian **mwlib**.

See Also

Library, mwlib

libm — Library

libm is the archive file that holds the mathematics library. For a summary of these routines, see **mathematics** and **extended mathematics**.

libm's table of contents can be printed and its contents altered with the archiver **mwlib**.

See Also

extended mathematics, Library, mathematics, math.h, mwlib, xmath.h

LIBPATH — Environmental variable

Directories that hold libraries

LIBPATH names the directories that **cc** searches to find the compiler's executable programs and libraries. **make** also searches these directories for the files **mmacros** and **mactions**.

For example, the command

```
set LIBPATH=\lib;\mwc
```

uses the MS-DOS command **set** to define **LIBPATH** as equalling **\lib;\mwc**. This definition of **LIBPATH** tells **cc** to look for the compiler's executable files first in directory **lib**, and then in directory **mwc**.

You may wish to write this command into the file **autoexec.bat**, so that **INCDIR** will be set automatically whenever you boot your system.

See Also

cc, environmental variables, make, PATH

limits.h — Header

The header **limits.h** defines a group of macros that set the numerical limits for the translation environment.

The following table gives the macros defined in **limits.h**. Each value given is the macro's minimum maximum: a conforming implementation of C must meet these limits, and may exceed them.

CHAR_BIT
: Number of bits in a **char**; must be at least eight.

CHAR_MAX
: Largest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MAX**; otherwise, it is equal to the value of the macro **UCHAR_MAX**.

CHAR_MIN
: Smallest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MIN**; otherwise, it is zero.

INT_MAX
: Largest value representable in an object of type **int**; it must be at least 32,767.

INT_MIN
: Smallest value representable in an object of type **int**; it must be at most -32,767.

LONG_MAX
: Largest value representable in an object of type **long int**; it must be at least 2,147,483,647.

LONG_MIN
: Smallest value representable in an object of type **long int**; it must be at most -2,147,483,647.

MB_LEN_MAX
: Largest number of bytes in any multibyte character, for any locale; it must be at least one.

SCHAR_MAX
: Largest value representable in an object of type **signed char**; it must be at least 127.

SCHAR_MIN
: Smallest value representable in an object of type **signed char**; it must be at most -127.

SHRT_MAX
: Largest value representable in an object of type **short int**; it must be at least 32,767.

SHRT_MIN
: Smallest value representable in an object of type **short int**; it must be at most -32,767.

UCHAR_MAX
: Largest value representable in an object of type **unsigned char**; it must be at least 255.

UINT_MAX
: Largest value representable in an object of type **unsigned int**; it must be at least 65,535.

ULONG_MAX
Largest value representable in an object of type **unsigned long int**; it must be at least 4,294,967,295.

USHRT_MAX
Largest value representable in an object of type **unsigned short int**; it must be at least 65,535.

Cross-references

Standard, §2.2.4.2
The C Programming Language, ed. 2, p. 257

See Also

Environment, header, numerical limits

link — Definition

To *link* a program means to resolve external references among individual source files. External references may refer to data or code that reside in another translation unit.

Some function calls may be resolved by the inclusion of the code for that function from a library, which consists of implementation-defined or user-defined functions.

See Also

compile, Definitions, linkage

linkage — Definition

The term *linkage* refers to the matching of an identifier with its namesakes across blocks of code, and among files of source code, pretranslated object modules, and libraries.

Identifiers can have internal linkage, external linkage, or no linkage. An identifier with external linkage is known across multiple translation units. An identifier with internal linkage is known only within one translation unit. An identifier with no linkage has no permanent storage allocated for it and is local to a function or block.

The following describes each type of linkage in more detail:

External linkage
The following identifiers have external linkage:

- Any identifier for a function that either has no storage-class identifier or is marked with the storage-class identifier **extern**, but excluding ones marked with the storage-class identifier **static**.
- Any global identifier that either has no storage-class identifier or is marked **extern**.

Internal linkage
The following identifiers have internal linkage:

- Any identifier marked **static**.
- Any identifier for a function that has file scope and is marked **static**.

No linkage
The following identifiers have no linkage:

- An identifier for anything that is not an object or function; e.g., a structure member, a **union** member, an enumeration constant, a tag, or a label.

- Any identifier declared to be a function parameter.
- An identifier local to a block (i.e., an **auto** object), that does not have file scope and is not marked **extern**.

An identifier with internal linkage may be up to at least 31 characters long, and may use both upper- and lower-case characters. An identifier with external linkage, however, may have up to at least six characters, and is not required to use both upper- and lower-case characters. These limits are implementation defined.

An object marked **extern** will have the same linkage as any previous declaration of the same object within that translation unit. If there is no previous declaration, the object has external linkage.

If an object appears in the same source file with external and internal linkage declarations, behavior is undefined. This is called a *linkage conflict*. It may occur if an object is first declared **extern**, then later re-declared to be **static**.

Cross-references

Standard, §3.1.2.2
The C Programming Language, ed. 2, p. 228

See Also

identifiers, name space, scope

locale.h — Header

Localization functions and macros
#include <locale.h>

locale.h is a header that declares or defines all functions and macros used to manipulate a program's locale. The Standard describes the following items within this header:

Type

lconv	Structure for numeric formatting

Manifest constants

LC_ALL	All locale information
LC_COLLATE	Locale collation information
LC_CTYPE	Locale character-handling information
LC_MONETARY	Locale monetary information
LC_NUMERIC	Locale numeric information
LC_TIME	Locale time information

Functions

localeconv	initialize **lconv** structure
setlocale	set/query locale

Cross-references

Standard, §4.4
The C Programming Language, ed. 2, pp

See Also

localization

localeconv() — Localization (libc)

Initialize **lconv** structure
#include <locale.h>
struct lconv *localeconv(void);

localeconv initializes the structure **lconv** and returns a pointer to it. **lconv** describes the formatting of numeric strings. For more information about this structure, see **lconv**.

The function **setlocale** establishes all or part of pre-defined locale as the current locale. A call to **setlocale** with the macros **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** may alter a portion of **lconv**.

Cross-reference

Standard, §4.4.2.1

See Also

lconv, localization, locale.h, setlocale

localization — Overview

The Standard introduced the concept of *localization* to C programming.

The Problem

C was originally designed to implement the UNIX operating system. As such, its formatting functions assumed that the Latin alphabet would be used (that is, the only characters 'a' through 'z' and 'A' through 'Z'), assumed that no accented characters would be required, and also assumed that numeric strings would be formatted as they are in the United States. Since its invention, however, C has grown out of its original setting and its original country: it is now used internationally to write a wide range of application software.

The Standard recognizes that C internally is based on the English language. That is, C's keywords and library names reflect its origin in English, and will continue to do so. Localization, however, allows an application program to use the character set and formatting information that is specific to a given country in certain aspects of the language.

A locale can be selected when the program is run, so applications can be user-selectable. It may include things like monetary formatting, but preserve the underlying data: only the presentation differs. Locales provide a standard way for software developers to use locale-specific information without having to "reinvent the wheel" for each locale.

If an implementation of C supports various locales, then that locale information need not be gathered by programmers who write applications software. Rather than each software house writing support for European collating conventions or Japanese monetary formatting conventions, the support is provided once, by the implementor, and in a standard fashion.

Locale Functions

The Standard describes two functions that can be used to access information specific to a given locale.

setlocale can be used in either of two ways: to set the current locale, or to query the current locale settings. Either part or all of a locale's strings can be set or queried.

localeconv initializes an instance of the structure **lconv** and returns a pointer to it. This structure holds information that can be used to print numeric and monetary strings. For more information on this structure, see the entry for **lconv**.

The macros that begin with **LC_** are defined in the header **locale.h**, and represent the categories of locales (also known as *locale strings*). The following describes the areas of C that are affected by

locales.

Characters
A national character set may include characters that lie outside of the Latin alphabet. Typically, these characters are not recognized as alphanumeric characters by functions like **isalpha**. To tell the translator to use the alternative character table for a given locale, use the call

```
setlocale(LC_CTYPE, locale);
```

The character-handling routines that are defined in the header **ctype.h** will use this locale information. This will also affect the functions that handle multibyte characters, as described below.

Collation
The sorting of strings that include national characters may present a problem. Normal collation functions depend upon the ASCII character order, and therefore do not know where additional, locale-specific characters go within the national character set. The Standard describes two functions, **strcoll** and **strxfrm**, that may collate strings which contain locale-specific characters. To set the locale information needed by these functions (so they know which national character order is used), use the call

```
setlocale(LC_COLL, locale);
```

strcoll and **strxfrm** will work in accordance with the current locale setting.

Date and Time
Most countries have an idiomatic way to express the current date and time. To set the locale information needed by the function **strftime**, use the call:

```
setlocale(LC_TIME, locale);
```

strftime can read the locale and format date and time strings accordingly.

Decimal Point
Different countries may use different characters to mark the decimal point. Occasionally, one character is used to mark the point in a numeric string and another to mark it in a string that describes money. The structure **lconv** contains the field **decimal_point**, which points to the character used to mark the decimal point in a numeric string.

To set the locale for functions that read or print the decimal point, use the call:

```
setlocale(LC_NUMERIC, locale);
```

All functions that perform string conversion, formatted output, or formatted input must interpret this information so these characters will be handled properly.

Money Each country has its own way to format monetary values. The character that represents the national currency varies from country to country, as does such aspects as whether the symbol goes before or after the numerals, how a negative value is rendered, what character is used to express a monetary decimal point (it may not be the same as the numeric decimal point), and how many digits are normally printed after the decimal point.

To set the locale information for money, use the call:

```
setlocale(LC_MONETARY, locale);
```

The structure **lconv**, which is initialized by the function **localeconv**, holds information needed to render monetary strings correctly.

LEXICON

Multibyte characters

Many countries, e.g., Japan and China, use systems of writing that use more characters than can be represented within one byte. Many operating systems and terminal devices, however, can receive only seven or eight bits at a time. To skirt this problem, the Standard describes two ways to encode such extensive sets of characters: with *wide characters* and *multibyte characters*.

A wide character is of type **wchar_t**. This type, in turn, is defined as being equivalent to the integral type that can describe all of the unique characters in the character set. This type is used mainly to store such characters in a device-independent manner.

A multibyte character, on the other hand, consists of two or more **char**s that together are understood by the terminal device as forming a non-alphabetic character or symbol. One wide character may map out to any number of multibyte characters, depending upon the number of systems of multibyte characters that are commonly in use.

The Standard describes five functions that manipulate wide characters and multibyte characters: **mblen**, **mbstowcs**, **mbtowc**, **wcstombs**, and **wctomb**. The actions of these functions are determined by the locale, as set by **setlocale**. To set a locale for the manipulation of multibyte characters, use the following call:

```
setlocale(LC_CTYPE, locale);
```

The Standard does not describe the mechanism by which tables of multibyte characters are made available to these functions.

Thousands

Large numbers can be broken up into groups of thousands to make them easier to read. The manner of grouping, including the number of items in each group and the character used to indicate the start of a new group, is locale specific.

The structure **lconv**, which is initialized by the function **localeconv**, contains the fields **thousands_sep**, **mon_grouping**, and **grouping**, which hold this information.

Default Locale

The only locale required of all conforming implementations is the **C** locale. This is the minimum set of locale strings needed to translate C source code. For a listing of what constitutes the **C** locale, see **lconv**.

When a C program begins, it behaves as if the call

```
setlocale(LC_ALL, "C");
```

had been issued.

Mechanism for Setting Locales

The Standard does not describe the mechanism by which **setlocale** makes locale information available to other functions, and by which the other functions use locale information. It is left to the implementation.

Cross-reference

Standard, §4.4

See Also

compliance, lconv, Library, locale.h

Notes

The Standard's section on compliance states that any program that uses locale-specific information does not conform strictly to the Standard. Therefore, a program that uses any locale other than the

C locale is not strictly conforming. A programmer should not count on being able to port such a program to any other implementation or execution environment.

localtime() — Time function (libc)

Convert calendar time to local time
#include <time.h>
struct tm ***localtime(const time_t** **timeptr*);**

localtime takes the calendar time pointed to by *timeptr* and breaks it down into a structure of type **tm**. Unlike the related function **gmtime**, **localtime** preserves the local time of the system. This local time includes conversion to daylight savings time, if applicable. The daylight savings time flag indicates whether daylight savings time is now in effect, *not* whether it is in effect during some part of the year. Note, too, that the time zone is set by **localtime** every time the value returned by

```
getenv("TIMEZONE")
```

changes. See the entry for **TIMEZONE** for more information on how **Let's C** handles time zone settings.

localtime returns a pointer to the structure **tm** that it creates. This structure is defined in the header **time.h**.

Example

The following example recreates the function **asctime**.

```
#include <stdio.h>
#include <time.h>

char *month[12] = {
      "January", "February" "March", "April",
      "May", "June", "July", "August",
      "September", "October", "November", "December"
};

char *weekday[7] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday"
};

main()
{
      char buf[20];
      time_t tnum;
      struct tm *ts;
      int hour = 0;

      /* get time from system */
      time(&tnum);

      /* convert time to tm struct */
      ts=localtime(&tnum);

      if(ts->tm_hour==0)
            sprintf(buf,"12:%02d:%02d A.M.",
                  ts->tm_min, ts->tm_sec);
```

```
      else
            if(ts->tm_hour>=12) {
                  hour=ts->tm_hour-12;
                  if (hour==0)
                        hour=12;
                  sprintf(buf,"%02d:%02d:%02d P.M.",
                        hour, ts->tm_min,ts->tm_sec);

            } else
                  sprintf(buf,"%02d:%02d:%02d A.M.",
                        ts->tm_hour, ts->tm_min, ts->tm_sec);

      printf("\n%s %d %s 19%d %s\n",
            weekday[ts->tm_wday], ts->tm_mday,
            month[ts->tm_mon], ts->tm_year, buf);

      printf("Today is the %d day of 19%d\n",
            ts->tm_yday, ts->tm_year);

      if(ts->tm_isdst)
            printf("Daylight Saving Time is in effect.\n");
      else
            printf("Daylight Saving Time is not in effect.\n");

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.3.4
The C Programming Language, ed. 2, p. 256

See Also

asctime, ctime, date and time, gmtime, local time, strftime, TIMEZONE< tm

Notes

localtime returns a pointer to a statically allocated data area that is overwritten by each successive call.

log() — Mathematics (libm)

Compute natural logarithm
#include <math.h>
double log(double *z***);**

log computes and returns the natural (base *e*) logarithm of its argument *z*. It is the inverse of the function **exp**.

Handing **log** an argument less than zero triggers a domain error. Handing it an argument equal to zero triggers a range error.

Cross-references

Standard, §4.5.4.4
The C Programming Language, ed. 2, p. 251

See Also

exp, frexp, ldexp, log10, mathematics, modf

log10() — Mathematics (libm)

Compute common logarithm
#include <math.h>
double log10(double *z***);**

log10 computes and returns the common (base 10) logarithm of its argument *z*.

Handing **log10** an argument less than zero triggers a domain error. Handing it an argument equal to zero triggers a range error.

Cross-references

Standard, §4.5.4.5
The C Programming Language, ed. 2, p. 251

See Also

exp, frexp, ldexp, log, mathematics, modf

long double — Type

A **long double** is a data type that represents at least a double-precision floating-point number. It is defined as being at least as large as a **double**. In some environments, extra precision can be gained by representing values with it.

Like all floating-point numbers, a **long double** consists of one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *mantissa*, or the number upon which the exponent works. The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the fraction will be increased. The structure of a **long double** and the range of values that it can encode are set in the following macros, all of which are defined in the header **limits.h**:

LDBL_DIG
: This holds the number of decimal digits of precision. This must be at least ten.

LDBL_EPSILON
: Where *b* indicates the base of the exponent (default, two) and *p* indicates the precision (or number of base *b* digits in the mantissa), this macro holds the minimum positive floating-point number *x* such that 1.0 + *x* does not equal 1.0, *b*^1-*p*. This must be at least 1E-9.

LDBL_MAX
: This holds the maximum representable floating-point number. It must be at least 1E+37.

LDBL_MAX_EXP
: This is the maximum integer such that the base raised to its power minus one is a representable finite floating-point number. No value is given for this macro.

LDBL_MAX_10_EXP
: This holds the maximum integer such that ten raised to its power is within the range of representable finite floating-point numbers. It must be at least +37.

LDBL_MANT_DIG
: This gives the number of digits in the mantissa. No value is given for this macro.

LDBL_MIN
: This gives the minimum value encodable within a **long double**. This must be at least 1E-37.

LDBL_MIN_EXP
: This gives the minimum negative integer such that when the base is raised to that power minus one is a normalized floating-point number. No value is given for this macro.

LEXICON

LDBL_MIN_10_EXP
: This gives the minimum negative integer such that ten raised to that power is within the range of normalized floating-point numbers.

A **long double** constant is represented by the suffix **l** or **L** on a floating-point constant.

For information about common floating-point formats, see **float**.

Cross-references

Standard, §2.2.4.2, §3.1.2.4, §3.1.3.1, §3.5.2
The C Programming Language, ed. 2, p. 196

See Also

double, float, types

long int — Type

A **long int** is a signed integral type. This type can be no closer to zero than an **int**.

A **long int** can encode any number between **LONG_MIN** and **LONG_MAX**. These are macros that are defined in the header **limits.h**. They are, respectively, -2,147,483,647 and 2,147,483,647.

The types **long**, **signed long**, and **signed long int** are synonyms for **long int**.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

int, short int, types

longjmp() — Non-local jumps (libc)

Execute a non-local jump
#include <setjmp.h>
void longjmp(jmp_buf *environment***, int** *rval***);**

A call to **longjmp** restores the environment that the function **setjmp** had stored within the array **jmp_buf**. Execution then continues not at the point at which **longjmp** is called, but at the point at which **setjmp** was called.

environment is the environment that had been saved by an earlier call to **setjmp**. It is of type **jmp_buf**, which is defined in the header **setjmp.h**.

longjmp returns the value *rval* to the original call to **setjmp**, as if **setjmp** had just returned. **rval** must be a number other than zero; if it is zero, then **setjmp** will return one.

Cross-references

Standard, §4.6.2.1
The C Programming Language, ed. 2, p. 254

See Also

non-local jumps, setjmp

Notes

Many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **longjmp** and **setjmp** will result in the creation of mysterious and irreproducible bugs.

longjmp will work correctly "in the contexts of interrupts, signals and any of their associated

functions." Also, **longjmp**'s behavior is undefined if it is used from within a function called by signal received during the handling of a different signal.

Experience has shown that **longjmp** should not be used within an exception handler. The Standard does not guarantee that programs will work correctly when **longjmp** is used to exit interrupts and signals. Experience has shown that even if the **longjmp** terminates the signal handler and returns successfully to the context of the **setjmp**, the program can easily fail to complete the very next function call it attempts, usually because the signal interrupted an update of a non-atomic data structure. The Standard guarantees that the implementations of **setjmp**, **longjmp**, and **signal** will work together; it cannot make any promises about the interactions of these services with other library functions or with user code. *Caveat utilitor*.

lseek() — Extended function (libc)

Set read/write position
long lseek(short *fd***, short** *how***, long** *where***);**

lseek sets the file-position indicator for stream *fp*. this changes the point where the next read or write operation will occur.

fd is the file's file descriptor, which is returned by **open**.

where and *how* describe the new seek position. *where* gives the number of bytes that you wish to move the seek position. It is measured from the beginning of the file if *how* is zero, from the current seek position if *how* is set to one, or from the end of the file if *how* is set to two.

A successful call to **lseek** returns the new seek position. For example,

```
position = lseek(filename, 100L, 0);
```

moves the seek position 100 bytes past the beginning of the file; whereas

```
position = lseek(filename, 0L, 1);
```

merely returns the current seek position, and does not change the seek position at all. **lseek** returns **-1L** if an error occurs, such as seeking to a negative position.

lseek differs from its cousin **fseek** in that **lseek** is an MS-DOS call and uses a file descriptor, whereas **fseek** is a library function and uses a **FILE** pointer.

See Also

extended miscellaneous, fseek, ftell

Notes

MS-DOS writes data at a physical address corresponding to the seek address. Thus, if you seek 10,000 bytes past the current end of file and write a string, the string will be written 10,000 bytes past the old end of file, and all the intervening data will then be made part of the file.

Some operating systems, such as MS-DOS, set the displacement from the file descriptor in bytes; others, such as VAX VMS, set the displacement in sectors. If you want your programs to be fully portable, you should avoid handing an absolute value to **lseek**.

lseek is not described in the ANSI Standard. A program that uses it does not comply strictly with the Standard, and may not be portable to other compilers or environments.

lvalue — Definition

An *lvalue* designates an object in storage. An lvalue can be of any type, complete or incomplete, other than type **void**.

A *modifiable lvalue* is any lvalue that is *not* of the following types:

- An array type.
- An incomplete type.
- Any type qualified by **const**.
- A structure or **union** with a member whose type is qualified by **const**, or with a member that is a structure or **union** with a member that is so qualified.

Only a modifiable lvalue is permitted on the left side of an assignment statement.

An lvalue normally is converted to the value that is stored in the designated object. When this occurs, it ceases to be an lvalue. For some lvalues, however, this does *not* occur, as follows:

- Any array type.
- When the lvalue is the operand of the operators **sizeof**, unary **&**, **--**, or **++**.
- When the lvalue is the left operand of the **.** operator.
- When the lvalue is the left operand of any assignment operator.

An lvalue with an array type is normally converted to a pointer to the same type. The value of the pointer is the address of the first member of the array. The exceptions to this operation are as follows:

- When it is the operand of the operators **sizeof** or unary **&**.
- When it is a string literal that initializes an array of **char**.
- When it is a string literal of wide characters that initializes an array of **wchar_t**.

In addition to the restrictions listed above, the following are also *not* lvalues, and hence cannot appear on the left side of an assignment statement:

- String literals.
- Character constants.
- Numeric constants.

Cross-references

Standard, §3.2.2.1
The C Programming Language, ed. 2, p. 197

See Also

conversions, function designator, rvalue

Notes

The term itself originally came from the phrase *left value*; in an expression like

```
object = value;
```

the element to the left of the '=' is the object whose value is modified. Because the Standard distinguishes between lvalues and modifiable lvalues, it prefers to define lvalue as being a contraction of the phrase *locator value*.

M

main — Definition

main is the name of the function that is called when a program begins execution under a hosted environment. A program must have one function named **main.** This function is special not only because it marks the beginning of program execution, but because it is the only function that may be called with either zero arguments or two arguments:

```
int main(void) {  }
```

or

```
int main(int argc, char *argv[]) {  }
```

Let's C allows **main** to take three arguments. Programs that use more than two arguments to **main**, however, do not conform strictly to the Standard.

The two **standard** arguments to **main** are called **argc** and **argv**. These names are used by convention; a programmer may use any names he wishes.

argv points to an array of pointers to strings. These strings can modify the operation of the program; thus, they are called *program parameters*. **argc** gives the number of strings in the array to which **argv** points.

The third variable to **main**, which is specific to **Let's C**, is **envp**. This variable points to an array of pointers to environmental variables.

If **main** calls **return**, it is equivalent to its calling **exit** with the same parameter. For example, the statement

```
return(EXIT_SUCCESS);
```

in **main** is equivalent to the call

```
exit(EXIT_SUCCESS);
```

If **main** returns without returning a value to the host environment, the value that is returned to the host environment is undefined.

Cross-references

Standard, §2.1.2.2
The C Programming Language, ed. 2, pp. 6, 164

See Also

argc, argv, Environment, envp

main — Technical information

Introduce program's main function

A C program consists of a set of functions, one of which must be called **main**. This function is called from the runtime startup routine after the runtime environment has been initialized.

Programs can terminate in one of two ways. The easiest is simply to have the **main** routine **return**. Control returns to the runtime startup; it closes all open file streams and otherwise cleans up, and then returns control to the operating system, passing it the value returned by **main** as exit status.

In some situations (errors, for example), it may be necessary to stop a program, and you may not want to return to **main**. Here, you can use **exit**; it cleans up the debris left by the broken program and returns control directly to the operating system.

A second exit routine, called **_exit**, quickly returns control to the operating system without performing any cleanup. This routine should be used with care, because bypassing the cleanup will leave files open and buffers of data in memory.

Programs compiled by **Let's C** return to the program that called them; if they return from **main** with a value or call **exit** with a value, that value is returned to their caller. Programs that invoke other programs through the **system** or **execall** functions check the returned value to see if these secondary programs terminated successfully.

See Also

argc, argv, envp, exit, _exit, runtime startup

make — Command

Program building discipline
make [*option …***] [***argument …***] [***target …***]**

make helps you build programs that consist of more than one file of source code.

Complex programs often consist of several *object modules*, each of which is the product of compiling a *source file*. A source file may refer to one or more **include** files, which can also be changed. Some programs may be generated from specifications given to program generators, such as **yacc**. Recompiling and relinking complicated programs can be difficult and tedious.

make regenerates programs automatically. It follows a specification of the structure of the program that you write into a file called **makefile**. **make** also checks the date and time that MS-DOS has recorded for each source file and its corresponding object module. To avoid unnecessary recompilation, **make** will recompile a source file only if it has been altered since its object module was last compiled.

The makefile

A **makefile** consists of three types of instructions: *macro definitions*, *dependency definitions*, and *commands*.

A macro definition simply defines a macro for use throughout the **makefile**. For example, the macro definition

```
FILES=file1.obj file2.obj file3.obj
```

The use of the equal sign '='.

A dependency definition names the object modules used to build the target program, and source files used to build each object module . It consists of the *target name*, or name of the program to be created, followed by a colon ':' and the names of the object modules that build it. For example, the statement

```
example:  $(FILES)
```

uses the macro **FILES** to name the object modules used to build the program **example**. Likewise, the dependency definition

```
file1.obj:  file1.c macros.h
```

defines the object module **file1.obj** as consisting of the source file **file1.c** and the header file **macros.h**.

Finally, a command line details an action that **make** must perform to build the target program. Each command line must begin with a space or tab character. For example, the command line

```
        cc -o example $(FILES)
```

LEXICON

gives the **cc** command needed to build the program **example**. The **cc** command lists the *object modules* to be used, *not* the source files.

Finally, you can embed comments within a **makefile**. **make** recognizes any line that begins with a pound sign '#' as being a comment, and ignores it.

make searches for **makefile** first in directories named in the environmental variable **PATH**, and then in the current directory.

Dependencies

The **makefile** specifies which files depend upon other files, and how to recreate the dependent files. For example, if the target file **test** depends upon the object module **test.obj**, the dependency is as follows:

```
test: test.obj
      cc -o test test.obj
```

make knows about common dependencies, e.g., that **.obj** files depend upon **.c** files with the same base name. The target **.SUFFIXES** contains the suffixes that **make** recognizes.

make also has a set of rules to regenerate dependent files. For example, for a source file with suffix **.c** and a dependent file with the suffix **.obj**, the target **.c.obj** gives the regeneration rule:

```
.c.obj:
      cc -c $<
```

The **-c** option to the **cc** commands tells **cc** not to link or erase the compiled object module. **$<** is a macro that **make** defines. It stands for the name of the file that causes the current action. The default suffixes and rules are kept in the files **mmacros** and **mactions**. The dependencies can be changed by editing these files.

Both of these must be kept in a directory named by the environmental variable **LIBPATH**. You can set this variable with the **set** command. For example, placing the command

```
set LIBPATH=\bin;\lib
```

into **autoexec.bat** sets **LIBPATH** to **\bin** and **\lib**. If **LIBPATH** is not set, the default directory is **\lib**.

Macros

To simplify the writing of complex dependencies, **make** provides a *macro* facility. To define a macro, write

NAME = string

The *string* is terminated by the end-of-line character, so it can contain blanks. To refer to the value of the macro, use a dollar sign '$' followed by the macro name enclosed in parentheses:

$(**NAME**)

If the macro name is one character, parentheses are not necessary. **make** uses macros in the definition of default rules:

```
.c.obj:
      $(CC) $(CFLAGS) -c $<
```

where the macros are defined as

```
CC=cc
CFLAGS=-V
```

The other built-in macros are:

$*	Target name, minus suffix
$@	Full target name
$<	List of referred files
$?	Referred files newer than target

Each command line *argument* should be a macro definition of the form

```
OBJECT=a.obj b.obj
```

You can override any built-in macro by resetting its value in the environment. For example, setting the following environmental variable

```
set CFLAGS=-VLARGE, -VCSD
```

ensures that **make** will always interpret the macro **CFLAGS** as meaning **-VLARGE,** regardless of how it is otherwise set in any file.

Options

The following lists the options that can be passed to **make** on its command line.

-d (Debug) Give verbose printout of all decisions and information going into decisions.

-f *file* *file* contains the **make** specification. If this option does not appear, **make** uses the file **makefile**, which is sought first in the directories named in the **PATH** environmental variable, and then in the current directory.

-i Ignore all errors from commands, and continue processing. Normally, **make** exits if a command returns an error.

-n Test only; suppresses actual execution of commands.

-p Print all macro definitions and target descriptions.

-q Return a zero exit status if the targets are up to date. Do not execute any commands.

-r Do not use the built-in rules that describe dependencies.

-s Do not print command lines when executing them. Commands preceded by '@' are not printed, except under the **-n** option.

-t (Touch option) Force the dates of targets to be the current time, and bypass actual regeneration.

See Also

as, cc, commands, ld

Diagnostics

make reports its exit status if it is interrupted or if an executed command returns error status. It replies "Target *name* not defined" or "Don't know how to make target *name*" if it cannot find appropriate rules.

Notes

The order of items in **mmacros\.SUFFIXES** is significant. The consequent of a default rule (e.g., **.obj**) must *precede* the antecedent (e.g., **.c**) in the entry **.SUFFIXES**. Otherwise, **make** will not work properly.

LEXICON

malloc() — General utility (libc)

Allocate dynamic memory
#include <stdlib.h>
void *malloc(size_t *size***);**

malloc allocates a block of memory *size* bytes long.

malloc uses a circular, first-fit algorithm to select an unused block of at least *size* bytes, marks the portion it uses, and returns a pointer to it. The function **free** returns allocated memory to the free memory pool.

Each area allocated by **malloc** is rounded up to the nearest even number and preceded by an **unsigned int** that contains the true length. Thus, if you ask for one byte, you will get four, and the **unsigned** that precedes the newly allocated area will be set to four.

When an area is freed, its low order bit is turned on. Consolidation occurs when **malloc** passes over an area as it searches for space. The end of each arena contains a block with a length of zero, followed by a pointer to the next arena. Arenas point in a circle.

The most common problem with **malloc** occurs when a program modifies more space than it allocates with **malloc**. This can cause later **malloc**s to go into a loop.

malloc returns a pointer to the block of memory it has allocated. The pointer is aligned for any type of object. If it could not allocate the amount of memory requested, it returns NULL.

Example

For an example of this function, see **realloc**.

Cross-references

Standard, §4.10.3.3
The C Programming Language, ed. 2, p. 167

See Also

__end, alignment, arena, calloc, free, general utilities, lmalloc, realloc

Notes

If *size* is set to zero, the behavior of **malloc** is implementation defined: **malloc** returns either NULL or a unique pointer. This is a quiet change that may silently break some existing code.

manifest constant — Definition

A *manifest constant* is a value that has been given a name.

The following demonstrates the definition of a manifest constant:

```
#define MAXFILES 9
```

Here, the constant **MAXFILES** is defined as having the value of nine. During the preprocessing phase of translation, the translator will substitute the character '9' for **MAXFILES** wherever it appears — or behave as if it had made such a substitution.

These constants serve two purposes within a C program: First, a constant can be changed throughout the program simply by changing its definition. Second, a programmer who reads the program will find it easier to understand the meaning of a well-named manifest constant than to understand its numeric analogue; for example, it is easy to grasp that **MAXFILES** represents the maximum number of files, but it is not nearly as easy to understand what **9** means.

Manifest constants have file scope, unless undefined with an **#undef** directive.

Cross-reference

The C Programming Language, ed. 2, p. 230

See Also

Definitions, macro, scope

Notes

The C Programming Language calls these constants *symbolic constants*.

math.h — Header

Header for mathematics functions
#include <math.h>

math.h is the header file that declares and defines mathematical functions and macros.

The Standard describes three manifest constants to be included in **math.h**, as follows:

EDOM	Domain error
ERANGE	Range error
HUGE_VAL	Unrepresentable object

The first two are used to set the global variable **errno** to an appropriate value when, respectively, a domain error or a range error occurs. **HUGE_VAL** is returned when any mathematics function attempts to calculate a number that is too large to be encoded into a **double**.

Let's C also includes 27 mathematics functions. For a listing of them, see **mathematics**.

Cross-references

Standard, §4.5
The C Programming Language, ed. 2, p. 250

See Also

header, mathematics

mathematics — Overview

The Standard describes 22 mathematics functions that are to be included with every conforming implementation of C, as follows:

60u

Exponent-log functions

exp	Compute exponential
frexp	Fracture floating-point number
ldexp	Load floating-point number
log	Compute natural logarithm
log10	Compute common logarithm
modf	Separate floating-point number

Hyperbolic functions

cosh	Calculate hyperbolic cosine
sinh	Calculate hyperbolic sine
tanh	Calculate hyperbolic tangent

Integer, value, remainder

ceil	Set integral ceiling of a number
fabs	Compute absolute value
floor	Set integral floor of a number
fmod	Calculate modulus for floating-point number

LEXICON

Power functions

pow	Raise one number to the power of another
sqrt	Calculate the square root of a number

Trigonometric functions

acos	Calculate inverse cosine
asin	Calculate inverse sine
atan	Calculate inverse tangent
atan2	Calculate inverse tangent
cos	Calculate cosine
sin	Calculate sine
tan	Calculate tangent

The Standard reserves all names that match those in this section and have a suffix of **f** or **l**, e.g., **ftan** or **ltan**. A future version of the Standard may provide additional library support for functions that manipulate **float**s or **long double**s.

Some existing implemetations may, on detection of domain or range errors, or other exceptional conditions, allow the function in question to call a user-specified exception handler, **matherr**. UNIX implementations have traditionally behaved this way. The Standard, in trying to accommodate a wide range of floating-point implementations, does not allow this behavior.

Cross-references

Standard, §4.5
The C Programming Language, ed. 2, p. 250

See Also

domain error, range error, Library, math.h

Notes

The Standard excludes the functions **ecvt**, **fcvt**, and **gcvt**, on the grounds that everything they do can be done more easily by the function **sprintf**.

maxmem — External data

extern unsigned int maxmem;
maxmem is an external variable that sets the maximum size of the program's data area. You can set **maxmem** in your program to protect a portion of memory from the memory allocation routine **sbrk**. Otherwise, **maxmem** is set to the end of physical memory by the C runtime startup routine.

See Also

__end, Environment, malloc, sbrk

mblen() — General utility (libc)

Return length of a string of multibyte characters
#include <stdlib.h>
int mblen(const char **address***, size_t** *number***);**

The function **mblen** checks to see if the *number* or fewer bytes of storage pointed to by *address* form a legitimate multibyte character. If they do, it returns the number of bytes that comprise that character. This function is equivalent to the call

```
mbtowc((wchar_t *)0, address, number);
```

If *address* is equivalent to NULL, then **mblen** returns zero if the current multibyte character set does not have state-dependent encodings and nonzero if it does. If *address* is not NULL, then **mblen** returns the following: (1) If *address* points to a null character, then **mblen** returns zero. (2)

If the *number* or fewer bytes pointed to by *address* forms a legitimate multibyte character, then **mblen** returns the number of bytes that comprise the character. (3) Finally, if the *number* bytes pointed to by *address* do not form a legitimate multibyte character, **mblen** returns -1. In no instance is the value returned by **mblen** greater than *number* or the value of the macro **MB_CUR_MAX**, whichever is less.

Cross-reference

Standard, §4.10.7.1

See Also

general utility, MB_CUR_MAX, mbtowc, wchar_t, wctomb

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

mbstowcs() — General utility (libc)

Convert sequence of multibyte characters to wide characters
#include <stdlib.h>
size_t mbstowcs(wchar_t **widechar***, const char** **multibyte***, size_t** *number***);**

The function **mbstowcs** converts a sequence of multibyte characters to their corresponding wide characters. It is the same as a series of calls of the type:

```
mbtowc(widechar, multibyte, MB_LEN_MAX);
```

except that the call to **mbstowcs** does not affect the internal state of **mbtowc**.

multibyte points to the base of the sequence of multibyte characters to be converted to wide characters. *widechars* points to the area where the converted characters are written, and *number* is the number of characters to convert. **mbstowcs** converts characters until either it reads a null character, or until it has converted *number* characters. In the latter case, then, no null character is written onto the end of the sequence of wide characters.

mbstowcs returns -1 cast to **size_t** if it encounters an invalid multibyte character before it has converted *number* multibyte characters. Otherwise, it returns the number of multibyte characters it converted to wide characters, excluding the null character that ends the sequence.

Cross-reference

Standard, §4.10.7.2

See Also

general utilities, wcstombs

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

mbtowc() — General utility (libc)

Convert a multibyte character to a wide character
#include <stdlib.h>
int mbtowc(wchar_t **charptr***, const char** **address***, size_t** *number***);**

The function **mbtowc** converts *number* or fewer bytes at *address* from a multibyte character to a wide character and stores the result in the area pointed to by *charptr*.

The behavior of **mbtowc** varies depending upon the values of *address* and *charptr*, as follows:

1. If *address* and *charptr* each point to a value other than NULL, then **mbtowc** reads the area pointed to by *address* and checks to see if *number* or fewer bytes comprise a legitimate multibyte character.

   If they do, then **mbtowc** stores the wide character that corresponds to that multibyte character in the area pointed to by *charptr* and returns the number of bytes that form the multibyte character.

   If *address* does not point to the beginning of a legitimate multibyte character, then **mbtowc** returns -1.

   Finally, if *address* points to a null character, **mbtowc** returns zero.

   In no instance does the value returned by **mbtowc** exceed *number* or value of the macro **MB_CUR_MAX**, whichever is less.

2. If *charptr* is set to NULL and *address* is set to a value other than NULL, then **mbtowc** behaves exactly like the function **mblen**: it examines the area pointed to by *address* but does not convert the multibyte character to a wide character.

3. If *address* is set to NULL, or both *address* and *charptr* are set to NULL, then **mbtowc** checks to see if the current multibyte character set have state-dependent encodings. **mbtowc** returns zero if the set does not have state-dependent encodings, and a number greater than zero if it does. It does not store anything in the area pointed to by *charptr*.

Cross-reference

Standard, §4.10.7.3

See Also

general utilities, MB_CUR_MAX, mblen, wchar_t, wctomb

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

me — Command

MicroEMACS screen editor
me [-e] [*file ...***]**

me is the command for MicroEMACS, the screen editor used by **Let's C**. With MicroEMACS, you can insert text, delete text, move text, search for a string and replace it, and perform many other editing tasks. MicroEMACS reads text from files and writes edited text to files. It can edit several files simultaneously, while displaying the contents of each file in its own screen window.

Screen Layout

If the command **me** is used without arguments, MicroEMACS opens an empty buffer. If used with one or more file name arguments, MicroEMACS will open each of the files named, and display its contents in a window. If a file cannot be found, MicroEMACS will assume that you are creating it for the first time, and create an appropriately named buffer and file descriptor for it.

The last line of the screen is used to print messages and inquiries. The rest of the screen is portioned into one or more *windows* in which MicroEMACS displays text. The last line of each window shows whether the text has been changed, the name of the buffer, and the name of the file associated with the window.

MicroEMACS notes its *current position*. It is important to remember that the current position is always to the *left* of the cursor, and lies *between* two letters, rather than at one letter or another.

For example, if the cursor is positioned at the letter 'k' of the phrase "Mark Williams", then the current position lies *between* the letters 'r' and 'k'.

Commands and Text

The printable ASCII characters, from **<space>** to '~', can be inserted at the current position. Control characters and escape sequences are recognized as *commands*, described below. A control character can be inserted into the text by prefixing it with **<ctrl-Q>** (that is, hold down the **<control>** key and type the letter 'Q').

There are two types of commands to remove text. *Delete* commands remove text and throw it away, whereas *kill* commands remove text but save it in the *kill buffer*. Successive kill commands append text to the previous kill buffer. Moving the cursor before you kill a line will empty the kill buffer, and write the line just killed into it.

Search commands prompt for a search string terminated by **<return>** and then search for it. Case sensitivity for searching can be toggled with the command **<esc>@**. Typing **<return>** instead of a search string tells MicroEMACS to use the previous search string.

Some commands manipulate words rather than characters. MicroEMACS defines a word as consisting of all alphabetic characters, plus '_' and '$'. Usually, a character command is a control character and the corresponding word command is an escape sequence. For example, **<ctrl-F>** moves forward one character and **<esc>F** moves forward one word. The MicroEMACS commands are not case sensitive. For example, **<ctrl-F>** and **<ctrl-f>** are identical.

Text can also be handled in blocks. MicroEMACS defines a block of text as all the text that lies between the *mark* and the current position of the cursor. For example, typing **<ctrl-W>** kills all text from the mark to the current position of the cursor. This is useful when moving text from one file to another. When you invoke MicroEMACS, the mark is set at the beginning of the file. You can reset the mark to the cursor's current position by typing **<ctrl-@>**.

Using MicroEMACS With the Compiler

MicroEMACS can be invoked automatically by the compiler command **cc** to help you repair all errors that occur during compilation. The **-A** option to **cc** causes MicroEMACS to be invoked automatically when an error occurs. The compiler error messages are displayed in one window, the source code in the other, and the cursor is at the line on which the first error occurred. When the text is altered, exiting from MicroEMACS automatically recompiles the file.

This cycle will continue either until the file compiles without error, or until you break the cycle by typing **<ctrl-U> <ctrl-X> <ctrl-C>**.

The option **-e** to the **me** command allows you to invoke the error buffer by hand.

The MicroEMACS Help Facility

MicroEMACS has a built-in help facility. With it, you can ask for information either for a word that you type in, or for a word over which the cursor is positioned. The MicroEMACS help file contains the bindings for all library functions and macros included with **Let's C**.

For example, consider that you are preparing a C program and want more information about the function **fopen**. Type **<ctrl-X>?**. At the bottom of the screen will appear the prompt

```
Topic:
```

Type **fopen**. MicroEMACS will search its help file, find its entry for **fopen**, then open a window and print the following:

```
Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (name, type) char *name, *type;
```

LEXICON

If you wish, you can kill the information in the help window and copy it into your program, to ensure that you prepare the function call correctly.

Consider, however, that you are checking a program written earlier, and you wish to check the call for a call to **fopen**. Simply move the cursor until it is positioned over one of the letters in **fopen**, then type **<esc>?**. MicroEMACS will open its help window, and show the same information it did above.

To erase the help window, type **<esc>2**.

Options

The following list gives the MicroEMACS commands. They are grouped by function, e.g., *Moving the cursor*. Some commands can take an *argument*, which specifies how often the command is to be executed. The default argument is 1. The command **<ctrl-U>** introduces an argument. By default, it sets the argument to four. Typing **<ctrl-U>** followed by a number sets the argument to that number. Typing **<ctrl-U>** followed by one or more **<ctrl-U>**s multiplies the argument by four.

Moving the Cursor

<ctrl-A> Move to start of line.

<ctrl-B> (Back) Move backward by characters.

<esc>B Move backward by words.

<ctrl-E> (End) Move to end of line.

<ctrl-F> (Forward) Move forward by characters.

<esc>F (Forward) Move forward by words.

<esc>G Go to an absolute line number in a file. Same as **<ctrl-X>G**.

<ctrl-N> (Next) Move to next line.

<ctrl-P> (Previous) Move to previous line.

<ctrl-V> Move forward by pages.

<esc>V Move backward by pages.

<ctrl-X>= Print the current position.

<ctrl-X>G Go to an absolute line number in a file. Can be used with an argument. Otherwise, it will prompt for a line number. Same as **<esc>G**.

<esc>! Move the current line to the line within the window given by *argument*. The position is in lines from the top if positive, in lines from the bottom if negative, and the center of the window if zero.

<esc>< Move to the beginning of the current buffer.

<esc>> Move to the end of the current buffer.

Killing and Deleting

<ctrl-D> (Delete) Delete next character.

<esc>D Kill the next word.

<ctrl-H> If no argument, delete previous character. Otherwise, kill *argument* previous characters.

<ctrl-K> (Kill) With no argument, kill from current position to end of line; if at the end, kill the newline. With argument set to one, kill from beginning of line to current position. Otherwise, kill *argument* lines forward (if positive) or backward (if negative).

<ctrl-W> Kill text from current position to mark.

<ctrl-X><ctrl-O>
Kill blank lines at current position.

<ctrl-Y> (Yank back) Copy the kill buffer into text at the current position. Set current position to the end of the new text.

<esc><ctrl-H>
Kill the previous word.

<esc><DEL>
Kill the previous word.

<DEL> If no argument, delete the previous character. Otherwise, kill *argument* previous characters.

Windows

<ctrl-X>1 Display only the current window.

<ctrl-X>2 Split the current window into two windows. This command is usually followed by **<ctrl-X>B** or **<ctrl-X><ctrl-V>**.

<ctrl-X>N (Next) Move to next window.

<ctrl-X>P (Previous) Move to previous window.

<ctrl-X>Z Enlarge the current window by *argument* lines.

<ctrl-X><ctrl-N>
Move text in current window down by *argument* lines.

<ctrl-X><ctrl-P>
Move text in current window up by *argument* lines.

<ctrl-X><ctrl-Z>
Shrink current window by *argument* lines.

Buffers

<ctrl-X>B (Buffer) Prompt for a buffer name, and display the buffer in the current window.

<ctrl-X>K (Kill) Prompt for a buffer name and delete it.

<ctrl-X><ctrl-B>
Display a window showing the change flag, size, buffer name, and file name of each buffer.

<ctrl-X><ctrl-F>
(File name) Prompt for a file name for current buffer.

<ctrl-X><ctrl-R>
(Read) Prompt for a file name, delete current buffer, and read the file.

LEXICON

<ctrl-X><ctrl-V>
(Visit) Prompt for a file name and display the file in the current window.

Saving Text and Exiting

<ctrl-X><ctrl-C>
Exit without saving text.

<ctrl-X><ctrl-S>
(Save) Save current buffer to the associated file.

<ctrl-X><ctrl-W>
(Write) Prompt for a file name and write the current buffer to it.

<ctrl-Z> Save current buffer to associated file and exit.

Compilation Error Handling

<ctrl-X>> Move to next error.

<ctrl-X>< Move to previous error.

Search and Replace

<ctrl-R> (Reverse) Incremental search backward. A pattern is sought as each character is typed.

<esc>R (Reverse) Search toward the beginning of the file. Waits for entire pattern before search begins.

<ctrl-S> (Search) Incremental search forward. A pattern is sought as each character is typed.

<esc>S (Search) Search toward the end of the file. Waits for entire pattern before search begins.

<esc>% Search and replace. Prompt for two strings, then search for the first string and replace it with the second.

<esc>/ Search for next occurrence of a string entered with the **<esc>S** or **<esc>R** commands. This remembers whether the previous search had been forward or backward.

<esc>@ Toggle case sensitivity for searches. By default, searches are case insensitive.

Keyboard Macros

<ctrl-X>(Begin a macro definition. MicroEMACS collects everything typed until the next **<ctrl-X>)** for subsequent repeated execution. **<ctrl-G>** breaks the definition.

<ctrl-X>) End a macro definition.

<ctrl-X>E (Execute) Execute the keyboard macro.

Change Case of Text

<esc>C (Capitalize) Capitalize the next word.

<ctrl-X><ctrl-L>
(Lower) Convert all text from current position to mark into lower case.

<esc>L (Lower) Convert the next word to lower case.

<ctrl-X><ctrl-U>
(Upper) Convert all text from current position to mark into upper case.

<esc>U (Upper) Convert the next word to upper case.

White Space

<ctrl-I> Insert a tab.

<ctrl-J> Insert a new line and indent to current level. This is often used in C programs to preserve the current level of indentation.

<ctrl-M> (Return) If the following line is not empty, insert a new line. If empty, move to next line.

<ctrl-O> Open a blank line; that is, insert newline after the current position.

<tab> With argument, set tab fields at every *argument* characters. An argument of zero restores the default of eight characters. Setting the tab to any character other than eight causes space characters to be set in your file instead of tab characters.

Send Commands to Operating System

<ctrl-C> Suspend MicroEMACS and pass commands to MS-DOS. Typing **exit** returns you to MicroEMACS and allows you to resume editing.

<ctrl-X>! Prompt for an MS-DOS command and execute it.

Setting the Mark

<ctrl-@> Set mark at current position.

<esc>. Set mark at current position.

<ctrl><space>
Set mark at current position.

Help Window

<ctrl-X>? Prompt for word for which information is needed.

<esc>? Search for word over which cursor is positioned.

<esc>2 Erase help window.

Miscellaneous

<ctrl-G> Abort a command.

<ctrl-L> Redraw the screen.

<ctrl-Q> (Quote) Insert the next character into text; used to insert control characters.

<esc>Q (Quote) Insert the next control character into the text. Same as **<ctrl-Q>**.

<ctrl-T> Transpose the characters before and after the current position.

<ctrl-U> Specify a numeric argument, as described above.

<ctrl-U><ctrl-X><ctrl-C>
Abort editing and re-compilation. Use this command to abort editing and return to MS-DOS when you are using the **-A** option to the **cc** command.

LEXICON

<ctrl-X>F Set word wrap to *argument* column. If argument is one, set word wrap to cursor's current position.

<ctrl-X><ctrl-X>
Mark the current position, then jump to the previous setting of the mark. This is useful when moving text from one place in a file to another.

MicroEMACS prints error messages on the bottom line of the screen. It prints informational messages (enclosed in square brackets '[' and ']' to distinguish them from error messages) in the same place.

MicroEMACS manipulates text in memory rather than in a file. The file on disk is not changed until you save the edited text. MicroEMACS prints a warning and prompts you whenever a command would cause it to lose changed text.

See Also

commands

Notes

Because MicroEMACS keeps text in memory, it does not work for extremely large files. It prints an error message if a file is too large to edit. If this happens when you first invoke a file, you should exit from the editor immediately. Otherwise, your file on disk will be truncated. If this happens in the middle of an editing session, however, delete text until the message disappears, then save your file and exit. Due to the way MicroEMACS works, saving a file after this error message has appeared will take more time than usual.

This version of MicroEMACS does not include many facilities available in the original EMACS display editor, which was written by Richard Stallman at M.I.T. In particular, it does not include user-defined commands or pattern search commands.

The current version of MicroEMACS, including source code, is proprietary to Mark Williams Company. The code may be altered or otherwise changed for your personal use, but it may *not* be used for commercial purposes, and it may not be distributed without prior written consent by Mark Williams Company.

MicroEMACS is based upon the public domain editor by David G. Conroy.

member — Definition

A *member* names an element within a structure or a **union**. It can be accessed via the member-selection operators '.' or '->'. For example, consider the following:

```
struct example {
      int member1;
      long member2;
      example *member3;
};

struct example object;
struct example *pointer = &object;
```

To read the contents of **member1** within **object**, use the '.', as follows:

```
object.member1
```

On the other hand, to read the contents of **member1** via **pointer**, use the '->' operator:

```
pointer->member1
```

The same is true for a **union**, but with the following restriction: if a value is stored in one member of a **union**, then attempting to read another member of the **union** generates implementation-defined

behavior. This restriction has one exception. If the **union** consists of several structures that have a common initial sequence, then that common sequence can be read when a value is written into any of the structures.

Cross-references

Standard, §3.1.2.6, §3.3.2.3
The C Programming Language, ed. 2, p. 128

See Also

->, ., name space, struct, union

memchr() — String handling (libc)

Search a region of memory for a character
#include <string.h>
void *memchr(const void **region***, int** *character***, size_t** *n***);**

memchr searches the first *n* characters in *region* for *character*. It returns a pointer to *character* if it is found, or NULL if it is not.

Unlike the string-search function **strchr**, **memchr** searches a region of memory. Therefore, it does not stop when it encounters a null character.

Example

The following example deals a random hand of cards from a standard deck of 52. The command line takes one argument, which indicates the size of the hand you want dealt. It uses an algorithm published by Bob Floyd in the September 1987 *Communications of the ACM*.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define DECK 52

main(int argc, char *argv[])
{
      char deck[DECK], *fp;
      int  deckp, n, j, t;

      if(argc != 2 ||
         52 < (n = atoi(argv[1])) ||
         1 > n) {
               printf("usage: memchr n # where 0 < n < 53\n");
            exit(EXIT_FAILURE);
      }

      /* exercise rand() to make it more random */
      srand((unsigned int)time(NULL));
      for(j = 0; j < 100; j++)
            rand();

      deckp = 0;
      /* Bob Floyd's algorithm */
      for(j = DECK - n; j < DECK; j++) {
            t = rand() % (j + 1);
            if((fp = memchr(deck, t, deckp)) != NULL)
                  *fp = (char)j;
            deck[deckp++] = (char)t;
      }
```

LEXICON

```
      for(t = j = 0; j < deckp; j++) {
            div_t card;

            card = div(deck[j], 13);
            t += printf("%c%c  ",
                  /* note useful string addressing */
                  "A23456789TJQK"[card.rem],
                  "HCDS"[card.quot]);

            if(t > 50) {
                  t = 0;
                  putchar('\n');
            }
      }

      putchar('\n');
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.1
The C Programming Language, ed. 2, p. 250

See Also

strchr, strcspn, string handling, strpbrk, strrchr, strspn, strstr, strtok

memcmp() — String handling (libc)

Compare two regions
#include <string.h>
int memcmp(const void ***region1***, const void** ***region2***, size_t** *count***);**

memcmp compares *region1* with *region2* character by character for *count* characters.

If every character in *region1* is identical to its corresponding character in *region2*, then **memcmp** returns zero. If it finds that a character in *region1* has a numeric value greater than that of the corresponding character in *region2*, then it returns a number greater than zero. If it finds that a character in *region1* has a numeric value less than less that of the corresponding character in *region2*, then it returns a number less than zero.

For example, consider the following code:

```
char region1[13], region2[13];
strcpy(region1, "Hello, world");
strcpy(region2, "Hello, World");
memcmp(region1, region2, 12);
```

memcmp scans through the two regions of memory, comparing **region1[0]** with **region2[0]**, and so on, until it finds two corresponding "slots" in the arrays whose contents differ. In the above example, this will occur when it compares **region1[7]** (which contains 'w') with **region2[7]** (which contains 'W'). It then compares the two letters to see which stands first in the character table used in this implementation, and returns the appropriate value.

Cross-references

Standard, §4.11.4.1
The C Programming Language, ed. 2, p. 250

See Also

strcmp, strcoll, string handling, strncmp, strxfrm

Notes

memcmp differs from the string comparison routine **strcmp** in the following ways:

First, **memcmp** compares regions of memory rather than strings; therefore, it does not stop when it encounters a null character.

Second, **memcmp** takes two pointers to **void**, whereas **strcmp** takes two pointers to **char**. The following code illustrates how this difference affects these functions:

```
char carray[10];
int iarray[10];
char *s = "hi";
   . . .
strcmp(carray, s)          /* RIGHT */
memcmp(carray, s, 3)       /* RIGHT */
strcmp(iarray, s)          /* WRONG, 1st arg not char * */
memcmp(iarray, s, 3)       /* RIGHT, args converted to void * */
```

It is wrong to use **strcmp** to compare an **int** array with a **char** array because this function compares strings. Using **memcmp** to compare an **int** array with a **char** array is permissible because **memcmp** simply compares areas of data.

memcpy() — String handling (libc)

Copy one region of memory into another
#include <string.h>
void *memcpy(void ******region1***, const void** ******region2***, size_t** *n***);**

memcpy copies *n* characters from *region2* into *region1*. Unlike the routines **strcpy** and **strncpy**, **memcpy** copies from one region to another. Therefore, it will not halt automatically when it encounters a null character.

memcpy returns *region1*.

Example

The following example copies a structure and displays it.

```
#include <string.h>
#include <stdio.h>

struct stuff {
      int a, b, c;
} x, y;

main(void)
{
      x.a = 1;
      /* this would stop strcpy or strncpy. */
      x.b = 0;
      x.c = 3;

      /* y = x; would do the same */
      memcpy(&y, &x, sizeof(y));
      printf("a =%d, b =%d, c =%d\n", y.a, y.b, y.c);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.2.1
The C Programming Language, ed. 2, p. 250

LEXICON

See Also

memmove, strcpy, string handling, strncpy

Notes

If *region1* and *region2* overlap, the behavior of **memcpy** is undefined. *region1* should point to enough reserved memory to hold *n* bytes of data; otherwise, code or data will be overwritten.

memmove() — String handling (libc)

Copy region of memory into area it overlaps
#include <string.h>
void *memmove(void ***region1*****, const void** ****region2*****, size_t** ***count*****);**

memmove copies *count* characters from *region2* into *region1*. Unlike **memcpy**, **memmove** correctly copies the region pointed to by *region2* into that pointed by *region1* even if they overlap. To "correctly copy" means that the overlap does not propagate, not that the moved data stay intact. Unlike the string-copying routines **strcpy** and **strncpy**, **memmove** continues to copy even if it encounters a null character.

memmove returns *region1*.

Example

The following example rotates a block of memory by one byte.

```
#include <string.h>
#include <stddef.h>
#include <stdio.h>

char *
rotate_left(char *region, size_t len)
{
      char sav;

      sav = *region;
      /* with memcpy this might propagate the last char */
      memmove(region, region + 1, --len);
      region[len] = sav;
      return(region);
}

char nums[] = "0123456789";
main(void)
{
      printf(rotate_left(nums, strlen(nums)));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.2.2 *The C Programming Language*, ed. 2, p. 250

See Also

memcpy, strcpy, string handling, strncpy

Notes

region1 should point to enough reserved memory to hold the contents of *region2*. Otherwise, code or data will be overwritten.

memset() — String handling (libc)

Fill an area with a character
#include <string.h>
void *memset(void ****buffer*, int** *character***, size_t** *n***);**

memset fills the first *n* bytes of the area pointed to by *buffer* with copies of *character*. It casts *character* to an **unsigned char** before filling *buffer* with copies of it.

memset returns the pointer *buffer*.

Example

The following example fills an area with 'X', and prints the result.

```
#include <stdio.h>
#include <string.h>
#define BUFSIZ 20

main(void)
{
      char buffer[BUFSIZ];

      /* fill buffer with 'X' */
      memset(buffer, 'X', BUFSIZ);

      /* append null to end of buffer */
      buffer[BUFSIZ-1] = '\0';

      /* print the result */
      printf("%s\n", buffer);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.6.1
The C Programming Language, ed. 2, p. 250

See Also

memchr, memcmp, memcpy, memmove, string handling

mktemp() — Extended function (libc)

Generate a temporary file name
char *mktemp(char ****pattern*);**

mktemp generates a unique file name. It can be used, for example, to name intermediate data files.

The *pattern* argument consists of a string that includes a capital '**X**'. **mktemp** replaces this **X** with '**A**' through '**Z**' to create up to 26 unique file names. It is normal practice to place temporary files in the directory **\tmp**. The start of the file name identifies the program that uses the file; for example, **\tmp\sortX** creates a temporary file to be used by **sort**. **mktemp** returns *pattern*.

The functions **tmpnam** and **tempnam** each assemble a temporary file name and then call **mktemp**. These routines ease the difficulty in creating a proper name for a temporary file.

See Also

extended miscellaneous, tempnam, tmpnam

LEXICON

mktime() — Time function (libc)

Turn broken-down time into calendar time

#include <time.h>
time_t mktime(struct tm ******timeptr***);**

mktime reads broken-down time from the structure pointed to by *timeptr* and converts it into calendar time of the type **time_t**. It does the opposite of the functions **localtime** and **gmtime**, which turn calendar time into broken-down time.

mktime manipulates the structure **tm** as follows:

1. It reads the contents of the structure, but ignores the fields **tm_wday** and **tm_yday**.
2. The original values of the other fields within the **tm** structure need not be restricted to the values described in the article for **tm**. This allows you, for example, to increment the member **tm_hour** to discover the calendar time one hour hence, even if that forces the value of **tm_hour** to be greater than 23, its normal limit.
3. When calculation is completed, the values of the fields within the **tm** structure are reset to within their normal limits to conform to the newly calculated calendar time. The value of **tm_mday** is not set until after the values of **tm_mon** and **tm_year**.
4. The calendar time is returned.

If the calendar time cannot be calculated, **mktime** returns -1 cast to **time_t**.

Example

This example gets the date from the user and writes it into a **tm** structure.

```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define BAD_TIME ((time_t)-1)

/* ask for a number and return it. */
int
askint(char * msg)
{
      char buf[20];

      printf("Enter %s ", msg);
      fflush(stdout);

      if(gets(buf) == NULL)
            exit(EXIT_SUCCESS);
      return(atoi(buf));
}

main(void)
{
      struct tm t;

      for(;;) {
            t.tm_mon  = askint("month");
            t.tm_mday = askint("day");
            t.tm_year = askint("year");
            t.tm_hour = t.tm_min = t.tm_sec = 1;
```

```
            if(BAD_TIME == mktime(&t)) {
                  printf("Invalid date\n");
                  continue;
            }

            printf("Day of week is %d\n", t.tm_wday);
            break;
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.2.3
The C Programming Language, ed. 2, p. 256

See Also

clock, date and time, difftime

Notes

The above description may appear to be needlessly complex. However, the Committee intended that **mktime** be used to implement a portable mechanism for determining time and for controlling time-dependent loops. This function is needed because not every environment describes time internally as a multiple of a known time unit.

model — Technical information

In the context of C programming, a **model** is a memory format that can be used on the i8086 microprocessor. Intel Corporation has defined six models for use on the i8086: TINY, SMALL, COMPACT, MEDIUM, LARGE, and HUGE. Mark Williams C compilers currently implement the SMALL and LARGE models, which experience shows can handle practically any programming task that can be executed with reasonable efficiency on the i8086.

In SMALL model, a program has two segments, each no larger than 64 kilobytes. One segment, the *code* segment, contains the code generated by the compiler. The other, the *data* segment, contains all pure and impure data, the stack, and the arena. "Pure" data are user data that have not yet been altered by the program, whereas "impure" data are user data that have been altered. In SMALL model, pointers are two **char**s (16 bits) long, which limits their addressing to 64 kilobytes.

In LARGE model, pointers consist of an offset and a segment. The actual address is calculated by shifting the segment left four and adding the offset. This can address up to one megabyte, although on the IBM PC the practical limit of memory is 640 kilobytes.

Code that is properly written can, in most instances, be ported from SMALL to LARGE model without modification. Routines that have integer-pointer puns, however, will run correctly under SMALL model but might fail under LARGE model.

See Also

LARGE model, pointer, pun, SMALL model, technical information

modf() — Mathematics (libm)

Separate floating-point number
#include <math.h>
double modf(double *real***, double** ******ip***);**

modf breaks the floating-point number *real* into its integer and fraction.

modf stores the integer in the location pointed to by *ip*, and returns the fraction *real*. Both the integer and the fraction have the same sign. *f* in the range 0 <= *f* < 1.

LEXICON

Cross-references

Standard, §4.5.4.6
The C Programming Language, ed. 2, p. 251

See Also

exp, frexp, ldexp, log, log10, mathematics, modf

mtype.h — Header

List processor code numbers

The header file **mtype.h** assigns a code number to each of the processors supported by Mark Williams C compilers. These include the Intel i8086, i8088, i80186, and i80286; the Zilog Z8001 and Z8002; the DEC PDP-11 and VAX; the IBM 370; and the Motorola 68000.

See Also

header, portability

multibyte characters — Overview

C was invented at Bell Laboratories as a portable language for implementing the UNIX operating system. Since then, C has grown into a language used throughout the world, for both operating systems and applications.

The character sets of many nations are too large to be encoded within one eight-bit byte. The Japanese Kanji characters form one such set; the ideograms of Mandarin Chinese form another. For the sake of brevity, the following discussion will call such sets *large-character sets*. A character from a large character set will be called a *large character*.

Wide Characters

The Standard describes two ways to encode a large character: by using a *multibyte character* or a *wide character*.

wchar_t is a typedef that is declared in the header **stdlib.h**. It is defined as the integral type that can represent all characters of given national character set.

The following restrictions apply to objects of this type: (1) The null character still has the value of zero. (2) The characters of the standard C character set must have the same value as they would when used in ordinary **char**s. (3) **EOF** must have a value that is distinct from every other character in the set.

wchar_t is a typedef of an integral type, whereas a multibyte character is a bundle of one or more one-byte characters. The format of a multibyte character is defined by the implementation, whereas a **wchar_t** can be used across implementations.

Wide characters are used to store large character sets in a device-independent manner. Multibyte characters are used most often to pass large characters to a terminal device. Most terminal devices can receive only one byte at a time. Thus, passing the pieces of a wide character to a terminal would undoubtedly create problems; the individual characters of a multibyte character, however, can be passed safely. This is also important because the Standard does not describe any function that reads more than one byte from a stream at any time — there is no Standard version of **fgetw** or **fputw**.

Multibyte Characters

The Standard describes multibyte characters as follows:

- A multibyte character may not contain a null character or 0xFF (-1, or **EOF**) as one of its bytes.
- All of the characters in the C character set must be present in any set of multibyte characters.
- An implementation of multibyte characters may use a *shift state* or a special sequence of characters that marks when a sequence of multibyte characters begins and when it ends. Depending upon the shift state, the bytes of a multibyte character may either be read as individual characters or as forming one multibyte character. Note, too, that a shift state may allow *state-dependent coding*, by which one of a number of possible sets of multibyte characters is indicated by the shift state.
- A comment, string literal, or character constant must begin and end in the same shift state. For example, a comment cannot consist of multibyte characters mixed with single-byte characters; it must be all one or all the other. If a comment, string literal, or character constant is built of multibyte characters, each such character must be valid.

Multibyte Character Functions

The support added to the C language for multibyte characters thus far is limited to character constants, string literals, and comments. The Standard describes five functions that handle multibyte characters:

mblen Compute length of a multibyte character
mbstowcs Convert sequence of multibyte characters to wide characters
mbtowc Convert multibyte character to wide character
wcstombs Convert sequence of wide characters to multibyte characters
wctomb Convert a wide character to a multibyte character

As mentioned above, a wide character is encoded using type **wchar_t**. The macro **MB_CUR_MAX** holds the largest number of characters of any multibyte character for the current locale. It is never greater than the value of the macro **MB_LEN_MAX**. **wcstombs** and **mbstowcs** convert sequences of characters from one type to the other.

All of the above are defined in the header **stdlib.h**.

Localization

The sets of multibyte characters and wide characters recognized by the above functions are determined by the program's locale, as set by the function **setlocale**.

To load the appropriate sets of multibyte characters and wide characters, use the call

```
setlocale(LC_CTYPE, locale);
```

or

```
setlocale(LC_ALL, locale);
```

See the entry for **localization** for more information.

Cross-reference

Standard, §2.2.1.2, §4.10.7

See Also

general utilities

Notes

Because compiler vendors are active in Asia, and because there is an active Japanese standards organization, a future version of the Standard may include more extensive support for multibyte characters, such as additional library functions. The support added to the C language for multibyte characters thus far is limited to character constants, string literals, and comments.

LEXICON

At present, all function names that begin with **wcs** are reserved. They should not be used if you wish your code to be maximally portable.

N

name space — Definition

The term *name space* refers to the "list" where the translator records an identifier. Each name space holds a different set of identifiers. If two identifiers are spelled exactly the same and appear within the same scope but are not in the same name space, they are *not* considered to be identical.

The four varieties of name space, as follows:

Label names
: The translator treats every identifier followed by a colon ':' or that follows a **goto** statement as a label.

Tags
: A tag is the name that follows the keywords **struct**, **union**, or **enum**. It names the type of object so declared.

Members
: A member names a field within a structure or a **union**. A member can be accessed via the operators '.' or '->'. Each structure or **union** type has a separate name space for its members.

Ordinary identifiers
: These name ordinary functions and variables. For example, the expression

```
int example;
```

: declares the ordinary identifier **example** to name an object of type **int**.

The Standard reserves external identifiers with leading underscores to the implementor. To reduce "name-space pollution," the implementor should not reserve anything that is not explicitly defined in the Standard (macros, **typedef**s, etc.) and that does not begin with a leading underscore.

Example

The following program illustrates the concept of name space. It shows how the identifier **foo** can be used numerous times within the same scope yet still be distinguished. This is extremely poor programming style. Please do not write programs like this.

```
#include <stdio.h>
#include <stdlib.h>

/* structure tag */
struct foo {
      /* structure member */
      struct foo *foo;
      int bar;
};

main(void)
{
      /* ordinary identifier */
      struct foo *foo;
      int i = 0;

      foo = (struct foo *)malloc(sizeof(foo));
      foo->bar = ++i;
      foo->foo = NULL;
```

```
/* label */
foo:  printf("Chain, chain, chain -- chain of \"foo\"s.\n");
      if (foo->foo == NULL) {
            foo->foo = (struct foo *)malloc(sizeof(foo));
            foo->foo->foo = NULL;
            foo->foo->bar = ++i;
            goto foo;
      }

      printf("The foo loop executed %d times\n", foo->foo->bar);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §3.1.2.3

See Also

identifiers, linkage, scope

Notes

Pre-ANSI implementations disagree on the name spaces of structure/**union** members. The Standard adopted the "Berkeley" rules, which state that every unique structure/**union** type has its own name space for its members. It rejected the rules of the first edition of *The C Programming Language*, which state that the members of all structures/**union**s reside in a common name space.

nested comments — Definition

Both *The C Programming Language*, ed. 2 and the draft ANSI standard declare that comments cannot be nested. Earlier versions of **Let's C** included a switch, called **-VCNEST**, that allowed a programmer to nest comments. This switch has been removed. Current and future versions of **Let's C** abort compilation when they detect nested comments.

See Also

Definitions, Language

nm — Command

Print a program's symbol table
nm [-**adgnopru**] *file* ...

nm prints the symbol table of each *file*. Each *file* argument must be a **Let's C** object module.

The first argument selects one of several options. It is optional; if present, it must begin with '-'. The options are as follows:

-a Print all symbols. Normally, **nm** prints names that are in C-style format and ignores symbols with names inaccessible from C programs.

-d Print only defined symbol.

-g Print only global symbols.

-n Sort numerically rather than alphabetically. **nm** uses unsigned compares when sorting symbols with this option.

-o Append the file name to the beginning of each output line.

-p Print symbols in the order in which they appear within the symbol table.

-r Sort in reverse-alphabetical order.

-u Print only undefined symbols.

By default, **nm** sorts symbol names alphabetically. Each symbol is followed by its value and its OMF segment.

See Also

cc, commands, ld, size, strip

nondigit — Definition

In the context of identifiers, a *nondigit* is any one of the following characters:

```
_     a     b     c     d     e     f     g     h
i     j     k     l     m     n     o     p     q
r     s     t     u     v     w     x     y     z
A     B     C     D     E     F     G     H     I
J     K     L     M     N     O     P     Q     R
S     T     U     V     W     X     Y     Z
```

Cross-reference

Standard, §3.1.2

See Also

digit, identifiers

non-local jumps — Overview

At times, exceptional conditions arise in a program that make it desirable to jump to a previous point within the program. **goto** can jump from one point to another within the same function, but it does not permit a jump from one function to another. The **setjmp/longjmp** mechanism was created to allow a program to jump immediately from one function to another, i.e., to perform a non-local jump.

The macro **setjmp** reads the machine environment and stores the environment in the array **jmp_buf**, which must be an array. The "machine environment" consists of the elements that determine the behavior of the machine, e.g., the contents of machine registers. What constitutes the machine environment will vary greatly from machine to machine. It may be impossible on some machines to save such elements of the machine environment as register variables and the contents of the stack or to restore the machine environment from within an extraordinarily complex computation.

For example, consider the following:

```
{
      int status[3][3][3], fn();
      jmp_buf buf;
      status[fn(1)][fn(2)][fn(3)] = setjmp(buf);
}
```

Here, the translator is trying to store the return value of **setjmp** into an array element with extremely complex index computations. It cannot be guaranteed that on every machine, the proper array element will be overwritten on reentry. For this reason, the Standard states that **setjmp** can be expected to save the machine environment only if used in a simple expression, such as in an **if** or **switch** statement.

The function **longjmp** jumps back to the point marked by the earlier invocation of **setjmp**. It restores the machine environment that **setjmp** had saved. This allows **longjmp** to perform a non-local jump.

LEXICON

A non-local jump can be dangerous. For example, many user-level routines cannot be interrupted and reentered safely. Thus, improper use of **longjmp** and **setjmp** with them will create mysterious and irreproducible bugs.

The Standard mandates that **longjmp** work correctly "in the contexts of interrupts, signals and any of their associated functions." Experience has shown, however, that **longjmp** should not be used within an exception handler that interrupts STDIO routines.

longjmp must not restore the machine environment of a routine that has already returned.

Cross-references

Standard, §4.6
The C Programming Language, ed. 2, p. 254

See Also

jmp_buf, Library, setjmp.h

Notes

longjmp's behavior is undefined if it is invoked from within a function that is called by a signal that is received during the handling of another signal. See **signal handling** for more information on signals.

notmem() — Extended function (libc)

Check if memory is allocated
int notmem(char ******ptr***);**

notmem checks if a memory block has been allocated by **calloc**, **malloc**, or **realloc**. *ptr* points to the block to be checked.

notmem searches the arena for *ptr*. It returns one if *ptr* is not a memory block obtained from **malloc**, **calloc**, or **realloc**, and zero if it is.

See Also

arena, calloc, extended miscellaneous, free, malloc, realloc, setbuf

null directive — Definition

Directive that does nothing

A *null directive* is a preprocessing directive that consists only of a '#' followed by **<newline>**. It does nothing.

Cross-reference

Standard, §3.8.7

See Also

preprocessing

null pointer constant — Definition

A *null pointer constant* is an integral constant expression with the value of zero, or such a constant that has been cast to type **void** *****. When the null pointer constant is compared with a pointer for equality, it is converted to the same type as the pointer before they are compared.

The null pointer constant always compares unequal to a pointer to an object or function. Two null pointers will always compare equal, regardless of any casts.

Cross-references

Standard, §3.2.2.3
The C Programming Language, ed. 2, p. 102

See Also

conversions, NULL

null statement — Definition

A *null statement* is one that consists only of a semicolon ';'. Its syntax is as follows:

null statement:
;

A null statement performs no operations.

Cross-references

Standard, §3.6.3
The C Programming Language, ed. 2, p. 222

See Also

Definitions, statements

numerical limits — Overview

The Standard describes numerical limits for every arithmetic type. For integral types, it sets the largest and smallest values that can be held in the given environment. For floating types, it also gives values for the manner in which a floating-point number is encoded.

These limits are recorded in two groups of macros: one for integral types, and the other for floating types. The groups of macros are kept, respectively, in the headers **limits.h** and **float.h**. The Lexicon entries for these headers lists the Standard's numerical limits.

Cross-references

Standard, §2.2.4.2
The C Programming Language, ed. 2, p. 257

See Also

Environment

Notes

The ANSI Committee has tried to keep its numerical limits compatible with those given in IEEE document 754, which describes a floating-point standard for binary number systems.

nybble — Definition

A **nybble** is four bits, or half of an eight-bit byte. The term is generally used to refer to the low four bits or the high four bits of a byte. Thus, a byte may be said to have a "low nybble" and a "high nybble". One nybble encodes one hexadecimal digit.

See Also

bit, byte, Definitions

O

object — Definition

An *object* is an area of memory that can contain one or more values. With the exception of a bit-field, an object consists of a byte or a contiguous group of bytes. The significance of each byte's value is defined by the program or the implementation. Objects that are variables are interpreted according to their type.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, p. 197

See Also

Definitions

object definition — Definition

A *definition* is a declaration that reserves storage for the thing declared. An *object definition* defines an object and makes it available throughout either the translation unit (if it has internal linkage) or throughout the program (if it has external linkage).

The term "tentative definition" refers to a definition to which more information is added by a later re-definition of the same object. The extra information may be a storage-class specifier, or it may initialize the object. The term, although somewhat misleading, is meant to show that every object has only one definition, but that definition can be refined during the course of translation.

Only one definition can contain an initializer. If an object is not initialized by the end of a file, it is initialized to zero.

A tentative definition of a static, incomplete object is disallowed semantically:

```
static int array[];
   . . .
int array[] = {3, 4, 5, 6};   /* Non-portable */
```

Because the Standard does not forbid an implementation to support such code, it may not generate an error message; however, this code is not portable.

The following *is* allowed semantically:

```
int array[];
   . . .
static int array[] = {3, 4, 5, 6};  /* RIGHT */
```

However, it may create a linker conflict in some implementations, such as in one-pass compilers.

To be assured that your code is maximally portable, declare the storage class and size of each object before you use it.

Cross-references

Standard, §3.7.2
The C Programming Language, ed. 2, p. 197

See Also

definition, external definitions, function definition, linkage, object

object format — Definition

An **object format** describes the form of compiled program that contains relocation information. The linker reads files in object format to create executable files.

See Also

Definitions, ld, n.out

object types — Definition

The *object types* are the set of types that describe objects. This set includes the **integral types**, the **floating types**, the **pointer types**, and the **aggregate types**.

Cross-reference

Standard, §3.1.2.5

See Also

function type, incomplete type, pointer, types

obsolescent — Definition

The term *obsolescent* refers to any feature of the C language that is widely used, but that may be withdrawn from future editions of the Standard. For example, consider the practice of first defining a function and then following the definition with a list of parameter declarations:

```
int example(parm1, parm2, parm3)
long parm1;
char *parm2;
int parm3;
{
      . . .
}
```

The Standard regards this as obsolete, and may eventually withdraw recognition of it in favor of the following syntax:

```
int example(long parm1, char *parm2, int parm3)
{
      . . .
}
```

The Standard regards three features of the language as being obsolete. The first is the use of separate lists of parameters identifiers and declaration lists, as described above. The second is the use of function declarators with empty parentheses; if a function takes no arguments, the word **void** should appear between the parentheses. The third is the placing of storage-class specifier at any point other than at the beginning of the declaration specifiers.

Cross-reference

Standard, §1.8, §3.9

See Also

Definitions, function declarators, storage-class specifiers

open() — Extended function (libc)

Open a file
short open(char **file***, short** *type***);**

LEXICON

open prepares a *file* to receive data, or to have its data read. When it can open *file*, **open** returns a file descriptor, which is a small, positive integer that identifies the opened *file* for subsequent calls to **read**, **write**, **close**, **dup**, or **dup2**.

type determines how the file is opened, as follows:

0	Read only
1	Write
2	Read and write

Once *file* is opened, reading or writing begins at byte 0.

open returns -1 if the file is nonexistent, or if a system resource is exhausted.

Example

This example copies the file named in **argv[1]** to the one named in **argv[2]**.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define BUFSIZE (20*512)
char buf[BUFSIZE];

/* prototypes for extended functions */
extern short close(short fd);
extern short open(char *file, short type);
extern short read(short fd, char *buffer, short n);
extern short write(short fd, char *buffer, short n);

void
fatal(char *message)
{
      fprintf(stderr, "copy: %s\n", message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      register short ifd, ofd;
      register unsigned short n;

      /* Check number of arguments */
      if (argc != 3)
            fatal("Usage: copy source destination");

      /* Open files */
      if ((ifd = open(argv[1], 0)) == -1)
            fatal("cannot open input file");
      if ((ofd = creat(argv[2], 0)) == -1)
            fatal("cannot open output file");

      /* Read and write text */
      while ((n = read(ifd, buf, BUFSIZE)) != 0) {
            if (n == -1)
                  fatal("read error");
            if (write(ofd, buf, n) != n)
                  fatal("write error");
      }
```

```
      if (close(ifd) == -1 || close(ofd) == -1)
            fatal("cannot close");
      exit(EXIT_SUCCESS);
}
```

See Also

close, extended miscellaneous, file descriptor, fopen

Notes

open is a low-level call that passes data directly to MS-DOS. It should not be mixed with high-level calls, such as **fread**, **fwrite**, or **fopen**.

open is not described in the ANSI Standard. Any program that uses it does not comply strictly with the Standard, and may not be portable to other compilers or environments.

operating system devices — Overview

Logical devices for system peripherals

MS-DOS gives names to its logical devices. **Let's C** uses these names to access these devices via MS-DOS.

MS-DOS includes the following logical devices:

aux	Auxiliary (serial) port
com1	Serial port
con	Console
lpt1	Parallel port; not always implemented
nul	Null device (the "bit bucket")
prn	Parallel port

You can use the function **fopen** to open these devices, just as if they were files. However, if you attempt to seek on a device, undefined behavior will occur.

See Also

Environment

operators — Overview

An *operator* specifies an operation performed upon one or two operands. The operation yields a value, performs designation, produces a side effect, or performs any combination of these.

The C language uses the following operators:

!	Not
!=	Compare two arithmetic operands for inequality
#	Substitute preprocessor token ("stringize")
##	Token-paste preprocessor tokens
%	Modulus operation on two arithmetic operands
%=	Modulus operation and assign result
&	Bitwise AND operation
&&	Logical AND for two expressions
&=	Bitwise AND operation and assign result
()	Cast operators
*	Multiply two arithmetic operands
***=**	Multiply two arithmetic operands and assign result
+	Add two arithmetic operands
++	Increment a scalar operand
+=	Add two operands and assign result

,	Evaluate an *rvalue*
-	Subtract two scalar operands, unary minus
--	Decrement a scalar operand
-=	Subtract two operands and assign result
->	Offset from structure/**union** pointer
.	Select member from structure/**union**
/	Divide two arithmetic operands
/=	Divide two arithmetic operands and assign result
<	Less than
<<	Bitwise left shift
<<=	Bitwise left shift and assign result
<=	Less than or equality
=	Assignment operator
==	Equality
>	Greater than
>=	Greater than or equal
>>	Bitwise right shift
>>=	Bitwise right shift and assign result
? :	Perform if/else operation
[]	Array subscript
^	Perform bitwise exclusive OR operation
^=	Perform bitwise exclusive OR and assign result
defined	Check if a macro is defined
sizeof	Size of operand in bytes
|	Perform bitwise OR operation
|=	Perform bitwise OR and assign result
||	Logical OR for two expressions
~	One's complement

The term *precedence* refers to the default order in which the operators in an expression are evaluated. The following list gives the default precedence of operators. Precedence is always overridden by the operators **()**, which, by default, enclose a primary expression:

Operator	*Associativity*
`() [] -> . (operand) * & sizeof`	Left to right
`~ ++ -- - * & sizeof`	Right to left
`* / %`	Left to right
`+ -`	Left to right
`<< >>`	Left to right
`< <= > >=`	Left to right
`== !=`	Left to right
`&`	Left to right
`^`	Left to right
`|`	Left to right
`&&`	Left to right
`||`	Left to right
`?:`	Right to left
`= += -= *= /= %=`	Right to left
`,`	Left to right

Cross-references

Standard, §3.1.5
The C Programming Language, ed. 2, pp. 41*ff*

See Also

lexical elements, punctuators

ordinary identifier — Definition

An *ordinary identifier* names all identifiers *except* labels, tags, and members. For example, the expression

```
int example;
```

declares the ordinary identifier **example** to name an object of type **int**.

Cross-references

Standard, §3.1.2.6
The C Programming Language, ed. 2, p. 192

See Also

name space

outb() — Extended function (libc)

Write to a port
int outb(int *port*, **int** *data*);

outb provides a C interface to the i8086 machine instruction **out**. It writes the least significant byte of the 16-bit word *data* to *port*, and returns *data*.

Example

For an example of this function, see the entry for **inb**.

See Also

extended miscellaneous, inb, inw, outw

P

parameter — Definition

The term *parameter* refers to an object that is declared with a function or a function-like macro.

With a function, a parameter is declared within a function declaration or definition. It acquires a value when the function is entered. For example, in the following declaration

```
FILE *fopen (const char *file, const char *mode);
```

file and **mode** are both objects that are declared within the function declaration. Both parameters will acquire their values when **fopen** is called.

With a function-like macro, a parameter is one of the identifiers that is bracketed by parantheses and separated by commas. For example, in the following example:

```
#define getchar(parameter) getc(stdin, parameter)
```

parameter is the identifier used with the macro **getchar**.

The scope of a function parameter is the block within which it is enclosed. The scope of a parameter to a function-like macro is the logical source line of the macro's definition.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, p. 202

See Also

argument, Definitions, function definition, scope

Notes

The Standard uses the term "argument" when it refers to the actual arguments of a function call or macro invocation. It uses the term "parameter" to refer to the formal parameters given in the definition of the function or macro.

PATH — Environmental variable

Directories that hold executable files

PATH names a default set of directories that are searched by MS-DOS when it seeks an executable file. You can set **PATH** with the MS-DOS command **path**. For example, typing

```
path c:\bin;a:\bin
```

tells MS-DOS to search for executable files first in **c:\bin**, and then in **a:\bin**.

For more information on the **path** command, see your MS-DOS user's manual.

See Also

environmental variable, path.h

path() — Access checking (libc)

Build a path name for a file
#include <path.h>
#include <stdio.h>
char *__path__**(char** ******path***, char** ******filename***, int** *mode***);**

The function **path** builds a path name for a file.

path points to the list of directories to be searched for the file. You can use the function **getenv** to obtain the current definition of the environmental variable **PATH**, or use the default setting of **PATH** found in the header file **path.h**, or, you can define *path* by hand.

filename is the name of the file for which **path** is to search. *mode* is the mode in which you wish to access the file, as follows:

1 Execute the file
2 Write to the file
4 Read the file

path uses the function **access** to check the access status of *filename*. If **path** finds the file you requested and the file is available in the mode that you requested, it returns a pointer to a static area in which it has built the appropriate path name. It returns NULL if either *path* or *filename* are NULL, if the search failed, or if the requested file is not available in the correct mode.

Example

This example accepts a file name and a search mode. It then tries to find the file in one of the directories named in the **PATH** environmental variable.

```
#include <path.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(char *message)
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      char *env, *pathname;
      int mode;

      if (argc != 3)
            fatal("Usage: findpath filename mode");

      if(((mode=atoi(argv[2]))>4) || (mode==3) || (mode<1))
            fatal("modes: 1=execute, 2=write, 3=read");

      env = getenv("PATH");
      if ((pathname = path(env, argv[1], mode)) != NULL) {
            printf("PATH = %s\n", env);
            printf("pathname = %s\n", pathname);
            return(EXIT_SUCCESS);
      } else
            fatal("search failed");
}
```

See Also

access, access checking, access.h, PATH, path.h

path.h — Header

Declare path()
#include <path.h>

path.h is a header that declares the function **path**. It also contains a number of default definitions for variables, including **PATH** and **LIBPATH**.

LEXICON

See Also

access checking, header, LIBPATH, path, PATH

pattern — Definition

A **pattern** is any combination of ASCII characters and wildcard characters that can be interpreted by a command.

The function **pnmatch** compares two patterns and signals if they match.

See Also

Definitions, egrep, pnmatch, wildcard

peek() — Extended function (libc)

Extract a word from memory
unsigned peek(unsigned *offs***, unsigned** *seg***);**

peek examines an arbitrary location in memory. It reads a word (two bytes) located at the offset *offs* and segment *seg*.

If your program is compiled into SMALL model, you can supply the offset/segment pair by using the macro **PTR**.

The header file **bios.h** declares a structure that defines the entire MS-DOS BIOS data area. You can use it to access an area within the BIOS data area for **peek**ing or **poke**ing.

Example

This example reads the address where the IBM PC stores the current memory size.

```
#include <stdio.h>
#include <stdlib.h>
#define MEMSIZELOC 0x13

main(void)
{
      extern unsigned peek();
      int size;

      size = (int)peek(MEMSIZELOC, 0x0);
      printf("Memory size = %d Kbytes\n",size);
      return EXIT_SUCCESS;
}
```

See Also

BIOS data area, bios.h, extended miscellaneous, peekb, poke, pokeb

peekb() — Extended function (libc)

Extract a byte from memory
unsigned peekb(unsigned *offs***, unsigned** *seg***);**

peekb examines an arbitrary location in memory. It reads a byte located at the offset *offs* and segment *seg*.

Note that if your program is compiled into SMALL model, you can supply the offset/segment pair by using the macro **PTR**.

Example

This example reads the MS-DOS location that holds the amount of memory on your machine.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      extern unsigned peekb();
      unsigned hbyte, lbyte, word;

      hbyte = peekb(0x14,0x0);
      lbyte = peekb(0x13,0x0);
      word = ((hbyte << 8) | lbyte);
      printf("Memory size = %d Kbytes\n",(int)word);
      return EXIT_SUCCESS;
}
```

See Also

_copy, csreg, extended miscellaneous, peek, poke, pokeb, PTR, _zero

perror() — STDIO (libc)

Write error message into standard error stream
#include <stdio.h>
void perror(const char ******string***);**

perror checks the integer expression **errno**, then writes the message associated with the value of **errno** into the standard error stream.

string points to a string that will prefix the error message, followed by a colon. For example, the call

```
perror("example");
```

ensures that the string

```
example:
```

will appear before any message that **perror** writes. If *string* is set to NULL, then the message will have no prefix.

Example

For an example of this function, see **feof**.

Cross-references

Standard, §4.9.10.4
The C Programming Language, ed. 2, p. 248

See Also

clearerr, errno, error codes, feof, ferror, STDIO, strerror

Notes

perror differs from the related function **strerror** in that it writes the error message directly into the standard error stream, instead of returning a pointer to the message.

The text of the message returned by **strerror** and the error-specific part of the message produced by **perror** should be the same for any given error number.

The external array **sys_errlist** gives the list of messages used by **perror**. The external variable **sys_nerr** gives the number of messages in the list.

picture() — Example

Format numbers under mask
double picture(double *number*, **const char** **mask, char* **output*);

picture uses a mask to format a double-precision number. It is designed to be used with programs that require precise formatting of printed numbers.

picture formats a given number by using a mask string. It writes its output into the area pointed to by *output*.

mask may contain any characters; however, only a few have special significance. Non-special characters in *mask* are printed if, during execution, they are preceded by one or more numerals. Trailing non-special characters print if the displayed number is negative.

picture returns all overflow as a **double**. For example, attempting to print **-1234** with mask **(ZZZ)** gives **(234)** and returns **-1**.

The following lists the special characters that control formatting within a mask:

9 Provides a slot for a number. For example, **5** with mask **999 CR** gives **005<sp><sp><sp>**, whereas printing **-5** with mask **999 CR** gives **005 CR**. 'C' and 'R' are not special characters, but are taken literally.

Z Provide a slot for a number but supress leading zeroes. For example, printing **1034** with mask **ZZZ,ZZZ** gives **<sp><sp>1,034**. The comma is not a special character, but is printed literally.

J Provide a slot for a number but shrink out leading zeroes. For example, printing **1034** with mask **JJJ,JJJ** gives **1,034**.

K Provide a slot for a number but shrink out all zeroes. For example, printing **070884** with mask **K9/K9/K9** gives **7/8/84**.

$ Print a dollar sign to the front of the displayed number. For example, printing **105** with mask **$Z,ZZZ** gives **<sp><sp>$105**.

. Separate the number between decimal and integer portions. For example, printing **105.67** with mask **ZZZ.999** gives **105.670**.

T Provide a slot for a number, but supress trailing zeroes. For example, printing **105.670** with mask **ZZ9.9TT** gives **105.67<sp>**.

S Provide a slot for a number, but shrink out trailing zeroes. For example, printing **105.600** with mask **ZZ9.9SS** gives **105.6**.

- If you place a hyphen to the left of the mask, it is printed at the beginning of the number, but only if it is negative. For example, printing **105** with mask **-Z,ZZZ** yields **<sp><sp>105**, whereas printing **-105** yields **<sp><sp>-105**.

(This character acts like the minus sign '-', but prints a '('. For example, printing **105** with mask **(ZZZ)** gives **<sp>105<sp>**, whereas printing **-5** gives **<sp><sp>(5)**.

+ If placed to the left of the mask, this character floats to the front like the minus sign '-', but is replaced by a '-' if the number is minus. For example, printing **5** with mask **+ZZZ** gives **<sp><sp>+5**, whereas printing **-5** gives **<sp><sp>-5**. Placed behind the mask, it is printed if the number is positive, but is replaced by a minus sign '-' if the number is negative. For example, printing **5** with mask **ZZZ+** gives **<sp><sp>5+**, whereas printing **-5** gives **<sp><sp>5-**.

* When placed to the left of the mask, this character fills all leading spaces to its right. For example, printing **104.10** with mask ***ZZZ,ZZZ.99** gives *****104.10**, and printing **104.10** with mask ***$ZZ,ZZZ.99** gives ****$104.10**.

Example

For an example of **picture**, compile the source program **picture.c** with the option **-DTEST**.

See Also

example

Notes

For the source code of **picture**, see the file **picture.c**, which is included with **Let's C**. **picture** is not included in a library.

pnmatch() — Extended function (libc)

Match string pattern
short pnmatch(char **string***, char** **pattern***, short** *flag***);**

pnmatch matches *string* with *pattern*, which is a regular expression.

pnmatch returns a positive number if *pattern* matches *string*, and zero if it does not.

Each character in *pattern* must exactly match a character in *string*. However, the wildcards '*', '?', '[', and ']' can be used in *pattern* to expand the range of matching. See **wildcards** for more information on what these symbols mean.

The *flag* argument must be either zero or one: zero means that *pattern* must match *string* exactly, whereas one means that *pattern* can match any part of *string*. In the latter case, the wildcards '^' and '$' can also be used in *pattern*.

Example

This example looks for the pattern given by **argv[1]** in standard input or in file **argv[2]**.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXLINE 128
char buf[MAXLINE];

void
fatal(char *message)
{
      fprintf(stderr, "pnmatch: %s\n", message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      /* Check that number of arguments is OK */
      if (argc != 2 && argc != 3)
            fatal("Usage: pnmatch pattern [ file ]");
      if (argc==3 && freopen(argv[2], "r", stdin)==NULL)
            fatal("cannot open input file");
```

```
      /* Get string, check with pattern */
      while (fgets(buf, MAXLINE, stdin) != NULL)
      {
            if (pnmatch(buf, argv[1], 1))
                  printf("%s", buf);
      }
      if (!feof(stdin))
            fatal("read error");
      exit(EXIT_SUCCESS);
}
```

See Also

extended miscellaneous, strcmp, strncmp, strstr, wildcards

Notes

flag must be zero or one for **pnmatch** to yield predictable results.

pnmatch is not described in the ANSI standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or other environments.

pointer — Definition

A *pointer* is an object whose value is the address of another object. The name "pointer" derives from the fact that its contents "point to" another object. A pointer may point to any type, complete or incomplete, including another pointer. It may also point to a function, or to nowhere.

The term *pointer type* refers to the object of a pointer. The object to which a pointer points is called the *referenced type*. For example, an **int** * ("pointer to **int**") is a pointer type; the referenced type is **int**. Constructing a pointer type from a referenced type is called *pointer type derivation*.

The Null Pointer

A pointer that points to nowhere is a *null pointer*. The macro **NULL**, which is defined in the header **stddef.h**, defines the null pointer for a given implementation. The null pointer is an integer constant with the value zero, or such a constant cast to the type **void** *. It compares unequal to a pointer to any object or function.

Declaring a Pointer

To declare a pointer, use the indirection operator '*'. For example, the declaration

```
int *pointer;
```

declares that the variable **pointer** holds the address of an **int**-length object.

Likewise, the declaration

```
int **pointer;
```

declares that **pointer** holds the address of a pointer whose contents, in turn, point to an **int**-length object. See **declarations** for more information.

Wild Pointers

Pointers are omnipresent in C. C also allows you to use a pointer to read or write the object to which the pointer points; this is called *pointer dereferencing*. Because a pointer can point to any place within memory, it is possible to write C code that generates unpredictable results, corrupts itself, or even obliterates the operating system if running in unprotected mode. A pointer that aims where it ought not is called a *wild pointer*.

When a program declares a pointer, space is set aside in memory for it. However, this space has not yet been filled with the address of an object. To fill a pointer with the address of the object you wish

to access is called *initializing* it. A wild pointer, as often as not, is one that is not properly initialized.

Normally, to initialize a pointer means to fill it with a meaningful address. For example, the following initializes a pointer:

```
int number;
int *pointer;
   . . .
pointer = &number;
```

The address operator '&' specifies that you want the address of an object rather than its contents. Thus, **pointer** is filled with the address of **number**, and it can now be used to access the contents of **number**.

The initialization of a string is somewhat different than the initialization of a pointer to an integer object. For example,

```
char *string = "This is a string."
```

declares that **string** is a pointer to a **char**. It then stores the string literal **This is a string** in memory and fills **string** with the address of its first character. **string** can then be passed to functions to access the string, or you can step through the string by incrementing **string** until its contents point to the null character at the end of the string.

Another way to initialize a pointer is to fill it with a value returned by a function that returns a pointer. For example, the code

```
extern void *malloc(size_t variable);
char *example;
   . . .
example = (char *)malloc(50);
```

uses the function **malloc** to allocate 50 bytes of dynamic memory and then initializes **example** to the address that **malloc** returns.

Reading What a Pointer Points To

The indirection operator '*' can be used to read the object to which a pointer points. For example,

```
int number;
int *pointer;
   . . .
pointer = &number;
   . . .
printf("%d\n", *pointer);
```

uses **pointer** to access the contents of **number**.

When a pointer points to a structure, the elements within the structure can be read by using the structure offset operator '->'. See the entry for **->** for more information.

Pointers to Functions

A pointer can also contain the address of a function. For example,

```
char *(*example)();
```

declares **example** to be a pointer to a function that returns a pointer to a **char**.

This declaration is quite different from:

```
char **different();
```

LEXICON

The latter declares that **different** is a function that returns a pointer to a pointer to a **char**.

The following demonstrates how to call a function via a pointer:

```
(*example)(arg1, arg2);
```

Here, the '*' takes the contents of the pointer, which in this case is the address of the function, and uses that address to pass to a function its list of arguments.

A pointer to a function can be passed to another function as an argument. The library functions **bsearch** and **qsort** both take function pointers as arguments. A program may also use of arrays of pointers to functions.

void *

void * is the generic pointer; it replaces **char** * in that role. A pointer may be cast to **void** * and then back to its original type without any change in its value. **void** * is also aligned for any type in the execution environment.

For more information on the use of the generic pointer, see **void**.

Pointer Conversion

One type of pointer may be converted, or *cast*, to another. For example, a pointer to a **char** may be cast to a pointer to an **int**, and vice versa.

Any pointer may be cast to type **void** * and back again without its value being affected in any way. Likewise, any pointer of a scalar type may be cast to its corresponding **const** or **volatile** version. The qualified pointers are equivalent to their unqualified originals.

Pointers to different data types are compatible in expressions, but only if they are cast appropriately. Using them without casting produces a *pointer-type mismatch*. The translator should produce a diagnostic message when it detects this condition.

Pointer Arithmetic

Arithmetic may be performed on all pointers to scalar types. Pointer arithmetic is quite limited and consists of the following:

1. One pointer may be subtracted from another.
2. An **int** or a **long**, either variable or constant, may be added to a pointer or subtracted from it.
3. The operators **++** or **--** may be used to increment or decrement a pointer.

No other pointer arithmetic is permitted.

Cross-references

Standard, §3.1.2.5, §3.2.2.1, §3.2.2.3, §3.3.2.2-3, §3.5.4.1
The C Programming Language, ed. 2, pp. 93*ff*

See Also

NULL, types, void

Notes

The Rationale cautions against using NULL as an explicit argument to any function that expects a pointer on the grounds that, under some environments, pointers to different data types may be of different lengths. All such problems will be avoided if a function prototype is within the scope of the function call. Then, NULL will be transformed automatically to the proper type of pointer. See *function prototype* for more information.

pointer declarators — Definition

A *pointer declarator* declares a pointer.

An asterisk '*' marks an identifier as being a pointer. For example:

```
int *example;
```

states that **example** is a pointer to **int**. Likewise, the use of two asterisks marks an identifier as being a pointer to a pointer. For instance,

```
int **example;
```

declares a pointer to a pointer to an **int**. It is sometimes helpful to read a C declarator backwards, i.e., from right to left, to decipher it.

A pointer declarator may be modified by the type qualifiers **const** or **volatile**. For example, the declarator

```
int *const example;
```

declares that **example** is a constant pointer to a variable value of type **int**, whereas the declaration

```
const int *example;
```

declares that **example** is a variable pointer to a constant integer value. The same syntax applies to **volatile**. The declaration

```
const int *const example;
```

declares a constant pointer to a constant **int**.

Cross-references

Standard, §3.5.4.1
The C Programming Language, ed. 2, p. 94

See Also

***, declarators, pointer**

poke() — Extended function (libc)

Insert a word into memory
unsigned poke(unsigned *offs***, unsigned** *seg***, unsigned** *data***);**

poke writes a word (two bytes) into an arbitrary location in memory. It writes the word *data* into the memory location given by the segment *seg* and offset *offs*, and returns *data*.

If your program is compiled into SMALL model, you can supply the full offset/segment pair by using the macro **PTR**. See its entry in the Lexicon for more information.

Example

This program will print a reverse video 'A' on an IBM-PC monochrome screen.

```
#include <stdlib.h>
main(void)
{
      /* '70' = reverse video, '41'= 'A' */
      poke(0x000, 0xB000, 0x7041);
      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, pokeb, PTR, _zero

Notes

Because memory is not protected on the i8086, be careful that you have the correct memory location when using **poke**.

pokeb() — Extended function (libc)

Insert a byte into memory
unsigned pokeb(unsigned *offs***, unsigned** *seg***, unsigned** *data***);**

pokeb writes a byte of data into an arbitrary location of memory. It writes *data* into the memory location given by the segment *seg* and offset *offs*, and returns *data*.

If your program is compiled into SMALL model, you can supply the full offset/segment pair by using the macro **PTR**. See its entry in the Lexicon for more information.

Example

This program will print a 'W' in the upper left-hand corner of an IBM-PC monochrome screen.

```
#include <stdlib.h>
main(void)
{
      pokeb(0x0000, 0xB000, 0x57);
      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, poke, PTR

Notes

Because memory is not protected on the i8086, be careful that you have the correct memory location when using **pokeb**.

pokeb is not described in the ANSI Standard. All programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

port — Definition

A **port** passes data to and receives data from a remote device.

See Also

aux, com1, fclose, FILE, fopen, lpt1, prn, stream

portability — Definition

The term *portability* refers to a program's ability to be translated and executed under more than one environment. The Standard is designed so that if you adhere to it strictly, you will, in the words of the Rationale, "have a 'fighting chance' to make powerful C programs that are also highly portable ...."

Although true portability is an ideal that is difficult to realize, you can take a number of practical steps to ensure that your code is portable:

- Do not assume that an integer and a pointer have the same size. Remember that undeclared functions are assumed to return an **int**.

- Do not write routines that depend on a particular order of code evaluation, particular byte ordering, or particular length of data types, except for those specified within the Standard.
- Do not write routines that play tricks with a machine's "magic characters". For example, writing a routine that depends on a file's ending with **<ctrl-Z>** instead of **EOF** ensures that that code can run only under operating systems that recognize this magic character.
- Always use constant such as **EOF** and make full use of **#define** statements.
- Use headers to hold all machine-dependent declarations and definitions.
- Declare everything explicitly. In particular, be sure to declare functions as **void** if they do not return a value. This avoids unforeseen problems with undefined return values.
- Do not assume that all varieties of pointer are the same or can point anywhere. On some machines, for example, a **char** * is longer than an **int** *. On others, a function pointer aims at a different space than does a data pointer.
- NULL should not be used as an explicit argument to any function that expects a pointer because, under some environments, pointers to different data types may be of different lengths. All such problems are avoided if a function prototype is within the scope of the function call. Then, NULL is transformed automatically to the proper type of pointer.
- Always exit or return explicitly from **main**, even when the program has run successfully to its end.
- **int** is the register size of the machine. Use **short** or **long** wherever size is a consideration.
- Inevitably, you will have code that is not 100% portable. Try to separate code that is machine-specific or operating-system specific into its own file.

Cross-reference

The C Programming Language, ed. 2, p. 3

See Also

behavior, Definitions

pow() — Mathematics (libm)

Raise one number to the power of another

#include <math.h>
double pow(double *z***, double** *x***);**

pow calculates and returns *z* raised to the power of *x*.

Cross-references

Standard, §4.5.5.1
The C Programming Language, ed. 2, p. 251

See Also

mathematics, sqrt

Notes

A domain error occurs if *z* equals zero, if *x* is less than or equal to zero, or if *z* is less than zero and *x* is not an integer.

pr — Command

Paginate and print files
pr [*options***] [***file …***]**

The command **pr** paginates each *file* and writes it into the standard output. The file name '-' means standard input. If no *file* is specified, **pr** reads the standard input.

On each page, **pr** writes a header that gives the date, file name, and page and line numbers. **pr** may be used with the following options.

+*n* Skip the first *n* pages of each input file.

-*n* Print the text in *n* columns. This is used to print out material that was typed in one or more columns.

-h *header*
Use *header* in place of the text name in the title. If *header* is more than one word long, it must be enclosed within quotation marks.

-l*n* Set the page length to *n* lines (default, 66).

-m Print the texts simultaneously in separate columns. Each text will be assigned an equal amount of width on the page. Any lines longer than that will be truncated. This is used to print several similar texts or listings simultaneously.

-n Number each line as it is printed.

-s*c* Separate each column by the character *c*. You can separate columns with a letter of the alphabet, a period, or an asterisk. Normally, each column is left justified in a fixed-width field.

-t Suppress the printing of the header on each page, as well as the header and footer space.

-w*n* Set the page width to *n* columns (default, 80). Text lines are truncated to fit the column width. The maximum width is 256 columns.

See Also

commands

preprocessing numbers — Definition

A *preprocessing number* is one of the intermediate lexical elements handled during translation phases 1 through 6. As semantic analysis occurs in translation phase 7, the set of valid preprocessing numbers forms a superset of valid C numeric tokens.

A preprocessing number is any floating constant or integer constant. A preprocessing number begins with either a digit or a period '.', and may consist of digits, letters, periods, and the character sequences **e+**, **e-**, **E+**, or **E-**.

Cross-reference

Standard, §3.1.8

See Also

lexical elements, preprocessing, token, translation phases

printf() — STDIO (libc)

Format and print text into the standard output stream
#include <stdio.h>
int printf(const char **format* **...);**

printf constructs a formatted string and writes it into the standard output stream.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how a particular type is to be converted into text.

Each conversion specification is introduced by the percent sign '%', and is followed, in order, by one or more of the following:

- A flag, which modifies the meaning of the conversion specification.
- An integer, which sets the minimum width of the field upon which the text is printed.
- A period and an integer, which sets the precision with which a number is printed.
- One of the following modifiers: **h**, **l**, or **L**. Their use is discussed below.
- Finally, a character that specifies the type of conversion to be performed. These are given below. This is the only element required after a '%'.

After *format* can come one or more arguments. There should be one argument for each conversion specification within *format* of the type appropriate to its conversion specifier. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *format* should be followed by three arguments, being, respectively, an **int**, a **long**, and a pointer to **char**.

If there are fewer arguments than conversion specifications, then **printf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding conversion specifier, then the behavior of **printf** is undefined.

If it writes the formatted string correctly, **printf** returns the number of characters written; otherwise, it returns a negative number. **printf** can generate a string that is up to at least 509 characters long.

The following sections describe in detail the elements of the conversion specification.

Conversion Specifiers

If *format* includes any conversion specifiers other than the ones shown below, the behavior is undefined. If a **union**, an aggregate, or a pointer to a **union** or an aggregate is used as an argument, behavior is undefined.

c Convert the **int** or **unsigned int** argument to a character.

d Convert the **int** argument to signed decimal notation.

D Convert the **long** argument to signed decimal. This specifier is not described in the Standard. Programs that use it do not comply strictly with the Standard, and may not be portable to other compilers or environments.

e Convert the **double** argument to exponential form. The format is

*[-]d.dddddd***e+/-***dd*

At least one digit always appears to the left of the decimal point and as many as *precision* digits to the right of it (default, six). If the precision is zero, then no decimal point is printed.

LEXICON

E Same as **e**, except that 'E' is used instead of 'e'.

f Convert the **double** argument to a string of the form

[-]d.dddddd

At least one digit always appears to the left of the decimal point, and as many as *precision* digits to the right of it (default, six). If the precision is zero, then no decimal point is printed.

g Convert the **double** argument to either of the formats **e** or **f**. The number of significant digits is equal to the precision set earlier in the conversion specification. Normally, this conversion selects conversion type **f**. It selects type **e** only if the exponent that results from such a conversion is either less than -4 or greater than the precision.

G Same as **g**, except that it selects between conversion types **E** and **f**.

i Same as **d**.

n This conversion specification takes a pointer to an integer, into which it writes the number of characters **printf** has generated to the current point within *format*. It does not affect the string **printf** generates.

o Convert the **int** argument to unsigned octal digits.

O Convert the **long** argument to unsigned octal. This specifier is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

p This conversion sequence takes a pointer to **void**. It translates the pointer into a set of characters and prints them. What it generates is defined by the implementation.

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char *** that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string. This is roughly equivalent to the library function **vprintf**.

This specifier is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

s Print the string to which the corresponding argument points; the argument must point to a C string. It prints either the number of characters set by the precision, or to the end of the string, whichever is less. If no precision is specified, then the entire string is printed.

u Convert the **int** argument to unsigned decimal digits.

U Convert the **long** argument to unsigned decimal. This specifier is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

x Convert the **int** argument to unsigned hexadecimal characters. The values 10, 11, 12, 13, 14, and 15 are represented, respectively, by 'a', 'b', 'c', 'd', and 'e'.

X Same as **x**, except that the values 10, 11, 12, 13, 14, and 15 are represented, respectively, by 'A', 'B', 'C', 'D', and 'E'. In previous releases of **Let's C**, this specifier converted a **long** to unsigned hexadecimal characters. This change was made to conform to the ANSI Standard, and may require that some code be rewritten.

The description of each conversion specifier assumes that it will be used with an argument whose type matches the type that the specifier expects. If the argument is of another type, it is cast to the type expected by the specifier. For example,

```
float f;
printf("%d\n", f);
```

will truncate **f** to an **int** before printing its value.

Flags

The '%' that introduces a conversion specification may be followed immediately by one or more of the following flags:

- Left-justify text within its field. The default is to right-justify all output text within its field.

+ Precede a signed number with a plus or minus sign. For example,

```
printf("%+d %+d\n", -123, 123);
```

yields the following when executed:

```
-123 +123
```

<space>
If the first character of a signed number is its sign, then that sign is appended to the beginning of the text string generated; if it is not a sign, then a space is appended to the beginning of the text string. For example,

```
printf("% d\n", -123);
printf("% d\n",  123);
```

generates the following:

```
-123
 123
```

This flag can be used with every conversion specifier for a numeric data type. It forces **printf** to use a special format that indicates what numeric type is being printed. The following gives the effect of this flag on each appropriate specifier:

e always retain decimal point
E always retain decimal point
f always retain decimal point
F always retain decimal point
g always retain decimal point; keep trailing zeroes
G always retain decimal point; keep trailing zeroes
x print '0x' before the number
X print '0X' before the number

Any specified precision is expanded by the appropriate amount to allow for the printing of the extra character or characters. Using '#' with any other conversion specifier yields undefined results.

0 When used with the conversion specifiers **d**, **e**, **E**, **f**, **g**, **G**, **i**, **o**, **u**, **x**, or **X**, a leading zero indicates that the field width is to be padded with leading zeroes instead of spaces. If precision is indicated with the specifiers **d**, **i**, **o**, **u**, **x**, **X**, then the **0** flag is ignored; it is also ignored if it is used with the **-** flag. If this flag is used with any conversion specifier other than the ones listed above, behavior is undefined.

Field Width

The field width is an integer that sets the minimum field upon which a formatted string is printed.

LEXICON

If a field width is specified, then that many characters-worth of space is reserved within the output string for that conversion. When the text produced by the conversion is *smaller* than the field width, spaces are appended to the beginning of the text to fill out the difference; this is called *padding*. Beginning the field width with a zero makes the padding character a '0' instead of a space. When the text is *larger* than the allotted field width, then the text is given extra space to allow it to be printed. Setting the field width never causes text to be truncated.

By default, text is set flush right within its field; using the '-' flag sets the text flush left within its field.

Using an asterisk '*' instead of an integer forces **printf** to use the corresponding argument as the field width. For example,

```
char *string = "Here's a number:";
int    width = 12;
int  integer = 123;
printf("%s%*d\n", string, width, integer);
```

produces the following text:

```
Here's a number:            123
```

Here, **width** was used to set the field width, so 12 spaces were used to pad the formatted integer.

Precision

The precision is indicated by a decimal point followed by a number. If a decimal point is used without a following number, then it is regarded as equivalent to '.0'.

The precision sets the number of characters to be printed for each conversion specifier. Setting the precision to *n* affects each conversion specifier as follows:

d print at least *n* digits
e print *n* digits after decimal point
E print *n* digits after decimal point
f print *n* digits after decimal point
g print no more than *n* significant digits
G print no more than *n* significant digits
i print at least *n* digits
o print at least *n* digits
s print no more than *n* characters
u print at least *n* digits
x print at least *n* digits
X print at least *n* digits

The precision differs from the field width in that the field width controls the amount of space set aside for the text, whereas the precision controls the number of characters to be printed. If the amount of padding called for by the precision conflicts with that called for by the field width, the amount called for by the precision is used.

Using an asterisk '*' instead of an integer forces **printf** to use the corresponding argument as the precision.

For example, this code

```
int   foo = 12345;
float bar = 12.345;
char *baz = "Hello, world";

printf("Example 1: %7.6d\n", foo);
printf("Example 2: %7.6f\n", bar);
printf("Example 3: %7.6s\n", baz);
```

produces the following text when executed:

```
Example 1:  012345
Example 2: 12.345000
Example 3:  Hello,
```

Modifiers

The following three modifiers may be used before a conversion specifier:

h When used before the specifiers **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument is a **short int** or an **unsigned short int**. When used before **n**, it indicates that the corresponding argument is a **short int**. In implementations where **short int** and **int** are synonymous, it is not needed; however, it is useful in writing portable code.

l When used before **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument is a **long int** or an **unsigned long int**. When used before 'n', it indicates that the corresponding argument is a **long int**. In implementations where **long int** and **int** are synonymous, it is not needed; however, it is useful in writing portable code.

L When used before **e**, **E**, **f**, **F**, or **G**, it indicates that the corresponding argument is a **long double**.

Using **h**, **l**, or **L** before a conversion specifier other than the ones mentioned above results in undefined behavior.

Default argument promotions are performed on the arguments. There is no way to suppress this.

Example

This example implements a mini-interpreter for **printf** statements. It is a convenient tool for seeing exactly how some of the **printf** options work. To use it, type a **printf** conversion specification at the prompt. The formatted string will then appear. To reuse a format identifier, simply type **<return>**.

```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* the replies go here */
static char reply[80];

/* ask for a string and echo it in reply. */
char *
askstr(char *msg)
{
      printf("Enter %s ", msg);
      fflush(stdout);

      if(gets(reply) == NULL)
            exit(EXIT_SUCCESS);
      return(reply);
}
```

LEXICON

```
main(void)
{
      char fid[80], c;

      /* initialize to an invalid format identifier */
      strcpy(fid, "%Z");

      for(;;) {
            askstr("format identifier");
            /* null reply uses previous FID */
            if(reply[0])
                  /* leave the '%' */
                  strcpy(fid + 1, reply);

            switch(c = fid[strlen(fid) - 1]) {
            case 'd':
            case 'i':
                  askstr("signed number");
                  if(strchr(fid, 'l') != NULL)
                        printf(fid, atol(reply));
                  else
                        printf(fid, atoi(reply));
                  break;

            case 'o':
            case 'u':
            case 'x':
            case 'X':
                  askstr("unsigned number");
                  if(strchr(fid, 'l') != NULL)
                        printf(fid, atol(reply));
                  else
                        printf(fid, (unsigned)atol(reply));
                  break;

            case 'f':
            case 'e':
            case 'E':
            case 'g':
            case 'G':
                  printf(fid, atof(askstr("real number")));
                  break;

            case 's':
                  printf(fid, askstr("string"));
                  break;

            case 'c':
                  printf(fid, *askstr("single character"));
                  break;

            case '%':
                  printf(fid);
                  break;

            case 'p':
                  /* print pointer to format id */
                  printf(fid, fid);
                  break;
```

```
            case 'n':
                  printf("n not implemented");
                  break;

            default:
                  printf("%c not valid", c);
            }

      printf("\n");
      }
}
```

Cross-references

Standard, §4.9.6.3
The C Programming Language, ed. 2, p. 244

See Also

fprintf, sprintf, STDIO, vfprintf, vprintf, vsprintf

Notes

printf can construct and output a string at least 509 characters long.

The character that **printf** prints to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

Because the **printf** routines that print floating-point numbers are quite large, they are included only optionally. If you wish to have **printf** print **float**s or **double**s, you must compile your program with the **-f** option to the **cc** command. See **cc** for more details.

prn — Operating system device

MS-DOS logical device for parallel port

MS-DOS gives names to its logical devices. **Let's C** uses these names, to allow the **STDIO** library routines to access these devices via MS-DOS.

prn is the logical device for the the parallel port.

Example

The following example checks to see if the parallel port can be opened.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      FILE *fp, *fopen();
      if ((fp = fopen("prn","w")) != NULL)
            fprintf(fp,"prn enabled.\n");
      else printf("prn cannot open.\n");
      return EXIT_SUCCESS;
}
```

See Also

aux, com1, con, lpt1, nul, operating system device

process — Definition

A **process** is a program in the state of execution.

See Also

daemon, Definitions, file

program startup — Definition

Program startup occurs when the execution environment invokes the program. Execution begins, and continues until program termination occurs. A program's execution may be suspended by the environment and resumed at a later time. The program, however, only starts once.

Cross-reference

Standard, §2.1.2

See Also

Environment, program termination

program termination — Definition

Program termination occurs when a program stops executing and returns control to the execution environment. Program termination may be triggered when the program calls either of the functions **abort** or **exit**, when **main** returns, when the environment or hardware raises a signal, or when program termination has been requested by some other program or event.

There are two types of termination: *unsuccessful* and *successful*.

Unsuccessful termination occurs either when a program aborts due to a significant problem in its operation (such as memory violation or division by zero), or when the program did not function as expected (such as when a requested file cannot be found).

A program indicates unsuccessful termination either by calling the function **exit** with the argument **EXIT_FAILURE**, by calling the function **abort**, or by using the function **raise** to generate the signal **SIGABRT**. **exit** is used to stop a program that cannot perform correctly, but does not threaten the integrity of the environment. **abort** and **raise** are used to stop a program that has gone seriously wrong.

Successful termination is declared to occur when the program runs to its conclusion correctly. A program indicates successful termination by calling the function **exit** with the argument **EXIT_SUCCESS**, or when **main** returns **EXIT_SUCCESS**.

Cross-reference

Standard, §2.1.2, §4.10.4.1, §4.10.4.3

See Also

abort, environment, execution environment, exit, EXIT_FAILURE, EXIT_SUCCESS, main, program startup, signal

pun — Definition

In the context of C, a **pun** occurs when a programmer uses one data form interchangeably with another. Puns are supported by C's willingness to apply implicit conversion rules.

A pun most often occurs unintentionally when the programmer fails to prototype or declare a function that returns a pointer. By default, the function is then assumed to return an **int**, and is handled as such. No trouble will arise if the program is run on a machine that defines an **int** and a pointer to have the same length (e.g., i8086 SMALL model); however, such code cannot be transported to an environment in which this is not the case (e.g., i8086 LARGE model).

See Also

Definitions, pointer, portability

punctuators — Overview

A *punctuator* is a symbol that has syntactic meaning but does not represent an operation that yields a value. All lexical elements that do not fall into another meaningful category are lumped together as punctuators.

Most often, a punctuator is used to mark or delimit an identifier or a portion of code, rather than modify it.

The set of punctuators consists of the following:

[]	Mark an array/delimit its size
()	Mark a parameter/argument list
{}	Delimit a block of code or a function
*	Identify a pointer type in a declaration
,	Delimit a function argument
:	Delimit a label
;	Mark end of a statement
...	(ellipsis) Indicate function takes flexible number of arguments
#	Indicate a preprocessor directive

The punctuators

```
{}   []   ()
```

must be used in pairs.

A symbol that acts as a punctuator may also act as an operator, depending upon its context.

Cross-reference

Standard, §3.1.6

See Also

lexical elements, operators, statements

putc() — STDIO (stdio.h)

Write a character into a stream
#include <stdio.h>
int putc(int *character*, **FILE** **fp*);

putc writes *character* into the stream pointed to by *fp*.

putc returns *character* if it was written correctly. Otherwise, it sets the error indicator for *fp* and returns **EOF**.

Example

This example writes newline characters into a file until the disk is full. Because this example uses the function **tmpfile**, the file it writes disappears when the program terminates. It is not recommended that you run this program on a multi-user system.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
```

LEXICON

```
main(void)
{
      long count;
      FILE *tmp;

      if((tmp = tmpfile()) == NULL) {
            fprintf(stderr, "Can't open tmp file\n");
            exit(EXIT_FAILURE);
      }

      for(count = 0; putc('\n', tmp) != EOF; count++)
            ;

      fprintf(stderr, "We wrote %ld characters\n", count);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.8
The C Programming Language, ed. 2, p. 247

See Also

getc, getchar, gets, putchar, puts, STDIO, ungetc

Notes

Because **putc** is implemented as a macro, *fp* may be read more than once. Therefore, one should beware of the side-effects of evaluating the argument more than once, especially if the argument itself has side-effects. See the entry for **macro** for more information. Use **fputc** if this behavior is not acceptable.

putchar() — STDIO (stdio.h)

Write a character into the standard output stream
#include <stdio.h>
int putchar(int *character***);**

putchar writes a character into the standard output stream. It is equivalent to:

```
putc(character, stdout);
```

putchar returns *character* if it was written correctly. If *character* could not be written, **putchar** sets the error indicator for the stream associated with stdout and returns **EOF**.

Example

This example prints all of the printable ASCII characters.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      char c;

      for(c = ' '; putchar(c) <= '}'; c++)
            ;

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.9
The C Programming Language, ed. 2, p. 247

See Also

getc, getchar, gets, puts, STDIO, ungetc

puts() — STDIO (libc)

Write a string into the standard output stream
#include <stdio.h>
int puts(char ******string***);**

puts replaces the null character at the end of *string* with a newline character, and writes the result into the standard output stream.

puts returns a non-negative number if it could write *string* correctly; otherwise, it returns **EOF**. In previous versions of **Let's C**, **puts** returned nothing. This was changed to conform to the ANSI Standard.

Example

This example uses **puts** to print a string into the standard output stream.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      puts("Hello world.");
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.10
The C Programming Language, ed. 2, p. 247

See Also

putc, putchar, STDIO, ungetc

Notes

For historical reasons, **fputs** writes *string* unchanged, whereas **puts** appends a newline character.

putw() — Extended macro (xstdio.h)

Write word to stream
#include <xstdio.h>
short putw(short *word***, FILE** ******fp***);**

The macro **putw** writes *word* onto the file stream *fp*. It returns the value written.

putw differs from **putc** in that **putw** writes an **int**, whereas **putc** writes a **char** that is promoted to an **int**.

putw returns **EOF** when an error occurs. You may need to call **ferror** to distinguish this value from a genuine end-of-file flag.

See Also

extended STDIO, ferror, xstdio.h

LEXICON

Notes

Because **putw** is implemented as a macro, arguments with side effects may not work as expected. The bytes of *word* are written in the natural byte order of the machine.

putw is not described in the ANSI Standard. A program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

The Standard requires that ANSI headers contain only functions that are described within the Standard. Therefore, **putw** has been moved from **stdio.h** to **xstdio.h**.

Q

qsort() — General utility (libc)

Sort an array

void qsort(void **array*, **size_t** *number*, **size_t** *size*, **int (****comparison*)
(const void **arg1*, **const void** **arg2*));

qsort sorts the elements within an array. *array* points to the base of the array being sorted; it has *number* members, each of which is *size* bytes long. In practice, *array* is usually an array of pointers and *size* is the **sizeof** the object to which each points.

comparison points to the function that compares two members of *array*. *arg1* and *arg2* each point to a member within *array*. The comparison routine must return a negative number, zero, or a positive number, depending upon whether *arg1* is, respectively, less than, equal to, or greater than *arg2*. If two or more members of *array* are identical, their ordering within the sorted array is unspecified.

Example

This example prints the command-line arguments in alphabetical order.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
compar(char *cp1[], char *cp2[])
{
      return(strcmp(*cp1, *cp2));
}

main(int argc, char *argv[])
{
      qsort((void *)++argv, (size_t)--argc, sizeof(*argv), compar);

      while(argc--)
            printf("%s ", *argv++);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.5.2
The C Programming Language, ed. 2, p. 87
The Art of Computer Programming, vol. 3

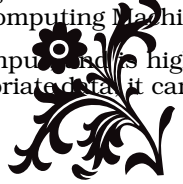
See Also

bsearch, general utilities

Notes

The name "qsort" reflects the fact that most implementations of this function (including **Let's C**) use C. A. R. Hoare's "quicksort" algorithm. This algorithm is recursive and makes heavy use of the stack. It is also specified by the Association for Computing Machinery's algorithm 271.

Quicksort works on the basis of partitioning its input, and is highly dependent on the first element that starts the partitioning process. Given appropriate data, it can have a worst-case performance of *O(n^2)*.



R

raise() — Signal handling (libc)

Send a signal
#include <signal.h>
int raise(int *signal***);**

raise sends *signal* to the program that is currently being executed. If called from within a signal handler, the processing of this signal may be deferred until the signal handler exits.

Example

This example sets a signal, raises it itself, then allows the signal to be raised interactivly. Finally, it clears the signal and exits.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void gotcha(void);

void
setgotcha(void)
{
      if(signal(SIGINT, gotcha) == SIG_ERR) {
            printf("Couldn't set signal\n");
            abort();
      }
}

void
gotcha(void)
{
      char buf[10];

      printf("Do you want to quit this program? <y/n> ");
      fflush(stdout);
      gets(buf);

      if(tolower(buf[0]) == 'y')
            abort();

      setgotcha();
}

main(void)
{
      char buf[80];

      setgotcha();
      printf("Set signal; let's pretend we get one.\n");
      raise(SIGINT);

      printf("Returned from signal\n");
      /* <ctrl-c> may not work on all operating systems */
      printf("Try typing <ctrl-c> to signal <enter> to exit");
      fflush(stdout);
      gets(buf);
```

```
      if(signal(SIGINT, SIG_DFL) == SIG_ERR) {
            printf("Couldn't lower signal\n");
            abort();
      }

      printf("Signal lowered\n");
      exit(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.7.2.1
The C Programming Language, ed. 2, p. 255

See Also

signal, signal handling, signal.h

Notes

This function is derived from the UNIX function **kill**.

rand() — General utility (libc)

Generate pseudo-random numbers
#include <stdlib.h>
int rand(void)

rand generates and returns a pseudo-random number. The number generated is in the range of zero to **RAND_MAX**, which equals 32,767.

rand will always return the same series of random numbers unless you change its *seed*, or beginning-point, with **srand**. Without having first called **srand**, it is as if you had initially set *seed* to one.

Example

This example produces a **char** that consists of random bits. The Standard's description of **rand** produces random **int**s, not random bits.

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

unsigned char
bitrand(void)
{
      register int i, r;

      for(i = r = 0; i < CHAR_BIT; i++) {
            r <<= 1;
            if(((long)rand() << 1) < (long)RAND_MAX)
                  r++;
      }
      return(r);
}

main(void)
{
      printf("Random stuff %02x %02x %02x\n",
            bitrand(), bitrand(), bitrand());
      return(EXIT_SUCCESS);
}
```

LEXICON

Cross-references

Standard, §4.10.2.2
The C Programming Language, ed. 2, p. 252

See Also

general utilities, RAND_MAX, srand

random access — Definition

In the context of computing, **random access** means that an entity can be accessed at any point, not just at the beginning. This means that all points within memory can be accessed equally quickly. This contrasts with *sequential access*, in which entities must be accessed in a particular order, so that some entities take longer to access than do others.

A tape drive is an example of a sequential access device, i.e., the order in which data are read is dictated by the order in which they stream past the tape head. Random-access memory (RAM) is an example of random access. Hard disks and floppy disks combine elements of random access and sequential access.

RAM, which usually consists of semiconductor integrated circuits, is also strictly random access. In this regard, the term "RAM" is slightly misleading; a more accurate name would be "read/write memory", to contrast RAM with read-only memory (ROM), which is also *random access* memory.

See Also

Definitions, read-only memory

read() — Extended function (libc)

Read from a file
short read(short *fd***, char** ***buffer***, short** *n***);**

read reads up to *n* bytes of data from the file descriptor *fd* and writes them into *buffer*. The amount of data actually read may be less than that requested if **read** detects EOF. The data are read beginning at the current seek position in the file, which was set by the most recently executed **read** or **lseek** routine. **read** advances the seek pointer by the number of characters read.

With a successful call, **read** returns the number of bytes read. Thus, zero bytes signals the end of the file. It returns -1 if an error occurs, such as bad file descriptor, bad *buffer* address, or physical read error.

Example

For an example of how to use this function, see the entry for **open**.

See Also

extended miscellaneous, fread

Notes

read is a low-level call that passes data directly to MS-DOS. It should not be intermixed with high-level calls, such as **fread**, **fwrite**, or **fopen**.

read is not described in the ANSI Standard. A program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

read-only memory — Definition

As its name suggests, **read-only memory,** or ROM, is memory that can be read but not overwritten. It most often is used to store material that is used frequently or in key situations, such as a language interpreter or a boot routine.

See Also

Definitions, random access

realloc() — General utility (libc)

Reallocate dynamic memory
#include <stdlib.h>
void *realloc(void ***_ptr_, size_t** _size_**);**

realloc reallocates a block of memory that had been allocated with the functions **calloc** or **malloc**. This function is often used to change the size of a block of allocated memory.

ptr points to the block of memory to reallocate. If *ptr* is set to NULL, then **realloc** behaves exactly the same as **malloc**: it allocates the requested amount of memory and returns a pointer to it. *size* is the new size of the block. If *size* is zero and *ptr* is not NULL, then the memory pointed to is freed.

realloc returns a pointer to the block of *size* bytes that it has reallocated. The pointer it returns is aligned for any type of object. If it cannot reallocate the memory, it returns NULL. It calls **abort** if it discoveres that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block.

Example

This example concatenates two strings that had been created with **malloc**.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *
combine(char **a, char **b)
{
      if(NULL == *a) {
            *a = *b;
            *b = NULL;
            return(*a);
      }
      else if(NULL == *b)
            return(*a);

      if((*a = realloc(*a, strlen(*a) + strlen(*b))) == NULL)
            return(NULL);
      return(strcat(*a, *b));
}

/* Copy a string into a malloc'ed hole. */
char *
copy(char *s)
{
      size_t len;
      char *ret;
```

```
      if(!(len = strlen(s)))
            return(NULL);
      if((ret = malloc(len)) == NULL)
            return(NULL);
      return(strcpy(ret, s));
}

main(void)
{
      char *a, *b;

      a = copy("A fine string. ");
      b = copy("Another fine string. ");

      puts(combine(&a, &b));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.3.4
The C Programming Language, ed. 2, p. 252

See Also

alignment, arena, calloc, free, general utility, lrealloc, malloc

Notes

If *size* is larger than the size of the block of memory that is currently allocated, the value of the pointer that **realloc** returns is indeterminate — it may point to the old block of memory, or it may not. If it is not, the contents of the old block of memory is copied to the new block.

record — Definition

A **record** is a set of data of a fixed length that has been given a unique identifier, and whose structure conforms to an exact description. An example of a record is an entry in a file of names and addresses: each entry has a fixed length, is marked by a unique identifier, and has a fixed number of bytes set aside in fixed order to record name, address, city, state, and ZIP code.

What is called a "record" in Pascal is called a "structure" in C.

See Also

Definitions, field, struct

register — C keyword

Quick access required
register *type identifier*

The storage-class specifier **register** declares that *identifier* is to be accessed as quickly as possible. **Let's C** will keep it in a machine register, if one is available.

It is not permissible to take the address of an object declared with the **register** designator, regardless of whether the implementation stores such an object in a machine register or not.

Example

For an example of using this specifier in a program, see **srand**.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, p. 83

See Also

storage-class identifiers

Notes

An implementation must document how it handles variables declared to be **register**. Practice currently ranges from ignoring register declarations completely, to allowing a few register declarations for objects of an appropriate type (typically integer or pointer), to ignoring the designator and implementing a full global register allocation scheme.

register — Definition

A **register** is special high-speed memory within a microprocessor that can be addressed concisely and within which data can be stored and modified. The size and the configuration of a microprocessor's registers affect its computing potential. Registers can be manipulated much faster than RAM.

The routines in the **Let's C** libraries generally assume that they have been called from C programs. Thus, they may freely overwrite any registers that the compiler overwrites in its generated code. Thus, for the i8086, a library routine that returns **int** returns its value in AX, and preserves SI, DI, BP; in SMALL model, it will also preserve DS and ES. It can freely overwrite BX, CX, DX; in LARGE model it will also overwrite DS and ES.

See Also

Definitions

remove() — STDIO (libc)

Remove a file
#include <stdio.h>
int remove(const char ***filename***);**

remove breaks the link between between *filename* and the actual file that it represents. In effect, it removes a file. Thereafter, any attempt to use *filename* to open that file will fail. It is equivalent to the function **unlink**.

If you attempt to remove a file that is currently open, **remove** will fail. **remove** returns zero if it could remove *filename*, and nonzero if it could not.

Example

This example removes the file named on the command line.

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
      if(argc != 1) {
            fprintf(stderr, "usage: remove filename\n");
            exit(EXIT_FAILURE);
      }

      if(remove(argv[1])) {
            perror("remove failed");
            exit(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

LEXICON

Cross-references

Standard, §4.9.4.1
The C Programming Language, ed. 2, p. 242

See Also

file operations, rename, tmpfile, tmpnam

rename() — STDIO (libc)

Rename a file
#include <stdio.h>
rename(const char ******old***; const char** ******new***);**

rename changes the name of a file, from the string pointed to by *old* to the string pointed to by *new*. Both *old* and *new* must point to a valid file name. If *new* points to the name of a file that already exists, the old file is replaced by the file being renamed.

rename returns zero if it could rename the file, and nonzero if it could not. If **rename** could not rename the file, its name remains unchanged.

Example

This example renames the file named in the first command-line argument to the name given in the second argument.

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
      if(argc != 3) {
            fprintf(stderr, "usage: rename from to\n");
            exit(EXIT_FAILURE);
      }

      if(rename(argv[1], argv[2])) {
            perror("rename failed");
            exit(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.4.2
The C Programming Language, ed. 2, p. 242

See Also

remove, STDIO, tmpfile, tmpnam

Notes

rename will fail if the file it is asked to rename is open, or if its contents must be copied in order to rename it.

return — C keyword

Return to calling function
return;
return *expression***;**

return is a statement that forces a function to return immediately to the function that called it.

return may also evaluate *expression* and pass its value to the calling function; the calling function regards this value as the value of the called function.

return can return a value to the calling function only if the called function was *not* declared to have a return type of **void**. The calling function is, of course, free to ignore the value **return** hands it.

If the called function is declared to return a type other than what **return** is actually returning, the value passed by **return** will be altered to conform to what the function was declared to return. For example,

```
main(void)
{
      printf("%s\n", example());
}

char *example(void)
{
      return "This is a string";
}
```

the pointer returned by **example** will be changed to an **int** before being returned to **main**. This is because **example** is declared implicitly within **main**, and a function that is declared implicitly is assumed to return an **int**. In environments where an **int** and a pointer are the same length, this code will work correctly. However, it will fail in environments where an **int** and a pointer have different lengths.

A function may have any number of **return** statements within it; however, a function can **return** only one value to the function that called it.

Reaching the last '}' in a function is equivalent to calling **return** without an expression.

Cross-references

Standard, §3.6.6.4
The C Programming Language, ed. 2, p. 70

See Also

break, C keywords, continue, goto, statements

Notes

If a program uses what is returned by a function as a value, and that function uses **return** without an expression, the behavior of the program is undefined.

rewind() — STDIO (libc)

Reset file-position indicator
#include <stdio.h>
void rewind(FILE ******fp***);**

rewind resets the file-position indicator to the beginning of the file associated with stream *fp*. It is equivalent to:

```
(void)fseek(fp, 0L, SEEK_SET);
```

rewind, unlike **fseek**, clears the error indicator for *fp*.

In previous releases of **Let's C**, **rewind** returned an **int**. It now returns nothing. This change was made to conform with the ANSI Standard, and may force some code to be rewritten.

LEXICON

Cross-references

Standard, §4.9.9.5
The C Programming Language, ed. 2, p. 248

See Also

fgetpos, file positioning, fseek, fsetpos, ftell

rindex() — Extended function (libc)

Find a character in a string
char ***rindex(char** **string*, **char** *character*);

rindex scans *string* for the last occurrence of *character*. If it finds *character*, it returns a pointer to it. If **rindex** does not find *character*, it returns NULL.

rindex is equivalent to the ANSI function **strchr**.

Example

This example uses **rindex** to help strip a sample file name of the path information. The path-name separator is '\'. The separator must be doubled so that **Let's C** will not interpret it as introducing an escape character.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define PATHSEP '\\'          /* path name separator */

char *
basename(char *path)
{
      char *cp;
      return (((cp = rindex(path, PATHSEP)) == NULL)
            ? path : ++cp);
}

main(void)
{
      char *testpath = "A:\\foo\\bar\\baz";

      printf("Before massaging: %s\n", testpath);
      printf("After massaging: %s\n", basename(testpath));
      return(EXIT_SUCCESS);
}
```

See Also

extended miscellaneous, index, memchr, strrchr

Notes

This function is identical to the ANSI function **strrchr**. It is recommended that you use **strrchr** instead of **rindex** so that your programs will more closely approach strict conformity with the Standard.

runtime startup — Overview

The C runtime startup is a routine that is linked with a C program as the first part of an executable program. It performs the tasks needed to start and terminate the C environment. To begin the program, it initializes the stack and calls **main**; to conclude the program, it calls **exit** with the return value from **main**.

Let's C includes the following runtime startup routines:

crts0xs.obj	SMALL model
crts0xl.obj	LARGE model

The runtime startups used with the option **-VCSD** generate executable files that can be debugged with the Mark Williams C source debugger **csd**.

All of the above routines call **_main**. Depending upon which options you use to the **cc** command, these routines in turn can call one or more of the following object modules:

csdxl.obj	LARGE model, debug information for **csd**
csdxs.obj	SMALL model, debug information for **csd**
fxl.obj	LARGE model, floating point/8087 sensing
fxs.obj	SMALL model, floating point/8087 sensing
fxl87.obj	LARGE model, floating point/8087 only
fxs87.obj	SMALL model, floating point/8087 only
naxl.obj	LARGE model, no arguments to command line
naxs.obj	SMALL model, no arguments to command line
nsxl.obj	LARGE model, no STDIO in executable
nsxs.obj	SMALL model, no STDIO in executable
wxl.obj	LARGE model, wildcards on command line
wxs.obj	SMALL model, wildcards on command line

See the Lexicon entry for **cc** for more information on the no STDIO (**-ns**) and wildcards (**-w**) options.

See Also

Environment, exargs, execall, function call, _main

Notes

Source code is included for some of the runtime startup routines. Note, however, that this code should be edited only by experienced systems programmers.

rvalue — Definition

An *rvalue* is the value of an expression. The name comes from the assignment expression **E1=E2;** in which the right operand is an rvalue.

Unlike an lvalue, an rvalue can be either a variable or a constant.

Although the term "rvalue" is commonly used among programmers, the Standard prefers the term "value of an expression".

Cross-references

The C Programming Language, ed. 2, pp

See Also

Definitions, lvalue

Notes

All non-void expressions have an rvalue.

LEXICON

S

sbrk() — Extended function (libc)

Increase a program's data space
char *sbrk(unsigned short *increment***);**

sbrk increases a program's data space by *increment* bytes. It increments the variable **__end**; this variable is set by the C runtime startup routine, and points to the end of the program's data space.

The memory-allocation function **malloc** calls **sbrk** should you attempt to allocate more space than is available in the program's data space.

sbrk returns a pointer to the previous setting of **__end** if the requested memory is available, or **((char *)-1)** if it is not.

See Also

__end, malloc, maxmem

Notes

sbrk will not increase the size of the program data area if the physical memory requested exceeds the physical memory allocated by MS-DOS, or if the requested memory exceeds the limit set in the user-defined variable **maxmem**. **sbrk** does not keep track of how space is used. Therefore, memory seized with **sbrk** cannot be freed. *Caveat utilitor*.

This function is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

scanf() — STDIO (libc)

Read and interpret text from standard input stream
#include <stdio.h>
int scanf(const char **format***, ...);*

scanf reads characters from the standard input stream and uses the string *format* to interpret what it has read into the appropriate types of data.

format is a string that consists of one or more conversion specifications, each of which describes how a portion of text is to be interpreted. *format* is followed by zero or more arguments. There should be one argument for each conversion specification within *format*, and each should point to the data type that corresponds to the conversion specifier within its corresponding conversion specification. For example, if *format* contains three conversion specifications that convert text into, respectively, an **int**, a **float**, and a string, then *format* should be followed by three arguments that point, respectively, to an **int**, a **float**, and an array of **char**s that is large enough to hold the string being input. If there are fewer arguments than conversion specifications, then **scanf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding type specification, then **scanf** returns.

scanf organizes the text read into a series of tokens. Each token is delimited by white space. White space usually is thrown away, except in the case of the 'c' or '[' conversion specifiers, which are described below.

If an input error occurs during input or if **EOF** is read, **scanf** returns immediately. If it reads an inappropriate character (e.g., an alphabetic character where it expects a digit), it returns immediately. **scanf** returns the number of conversions it accomplished. If it could accomplish no conversions, it returns **EOF**.

Conversion Specifications

The percent sign character '%' marks the beginning of a conversion specification. The '%' will be followed by one or more of the following:

- An asterisk '*', which tells **scanf** to skip the next conversion; that is, read the next token but do not write it into the corresponding argument.
- A decimal integer, which tells **scanf** the maximum width of the next field being read. How the field width is used varies among conversion specifier. See the table of specifiers below for more information.
- One of the three modifiers **h**, **l**, or **L**, whose use is described below.
- A conversion specifier, whose use is described below.

Modifiers

The following three modifiers may be used before a conversion specifier:

h When used before the conversion specifiers **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument points to a **short int** or an **unsigned short int**. When used before **n**, it indicates that the corresponding argument points to a **short int**. In implementations where **short int** and **int** are synonymous, it is not needed. However, it is useful in writing portable code.

l When used before the conversion specifiers **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument points to a **long int** or an **unsigned long int**. When used before **n**, it indicates that the corresponding argument points to a **long int**. In implementations where **long int** and **int** are synonymous, it is not needed. However, it is useful in writing portable code.

L When used before the conversion specifiers **e**, **E**, **f**, **F**, or **G**, it indicates that the corresponding argument points to a **long double**.

If **h**, **l**, or **L** is used before a conversion specifier other than the ones mentioned above, it is ignored. In previous releases of **Let's C**, the modifier **L** meant that the corresponding argument pointed to a **long** rather than to a **long double**, as it does now. This has been changed to conform to the ANSI Standard, and may require that some code be rewritten.

Conversion Specifiers

The Standard describes the following conversion specifiers:

c Convert into **char**s the number of characters specified by the field width, and write them into the array pointed to by the corresponding argument. The default field width is one. **scanf** does not write a null character at the end of the array it creates. This specifier forces **scanf** to read and store white-space characters and numerals, as well as letters.

d Convert the token to a decimal integer. The format should be equivalent to that expected by the function **strtol** with a base argument of ten. The corresponding argument should point to an **int**.

D Convert the token to a **long**. This conversion specifier is not described in the ANSI Standard, and using it means that your program will not comply strictly with the Standard.

e Convert the token to a floating-point number. The format of the token should be that expected by the function **strtod** for a floating-point number that uses exponential notation. The corresponding argument should point to a **double**.

LEXICON

E Same as **e**. Under earlier releases of **Let's C**, this conversion specifier converted the token to a **double**. This change has been made to conform to the ANSI Standard, and may require that some code be rewritten.

f Convert the token to a floating-point number. The format of the token should be that expected by the function **strtod** for a floating-point number that uses decimal notation. The corresponding argument should point to a **double**.

F Same as **f**.

g Convert the token to a floating-point number. The format of the token should of that expected by the function **strtod** for a floating-point number that uses either exponential notation or decimal notation. The corresponding argument should point to a **double**.

G Same as **g**.

i Convert the token to a decimal integer. The format should be equivalent to that expected by the function **strtol** with a base argument of zero. The corresponding argument should point to an **int**.

n Do not read any text. Write into the corresponding argument the number of characters that **scanf** has read up to this point. The corresponding argument should point to an **int**.

o Convert the token to an octal integer. The format should be equivalent to that expected by the function **strtol** with a base argument of eight. The corresponding argument should point to an **int**.

O Same as **o**, except that the corresponding argument points to a **long**. This conversion specifier is not described in the ANSI Standard, and using it means that your program will not comply strictly with the Standard.

p Pointer format: read a sequence of implementation-defined characters, convert them in an implementation-defined way, and write them in an implementation-defined manner. The vagueness of this description is unavoidable, because the pointer format will vary between machines, and even on the same machine. The corresponding argument should point to a **void** *. The sequence of characters recognized should be identical with that written by **printf**'s **p** conversion specifier.

s Read a string of non-white space characters, copy them into the area pointed to by the corresponding argument, and append a null character to the end. The argument should be of type **char** *, and should point to enough allocated memory to hold the string being read plus its terminating null character.

u Convert the token to an unsigned integer. The format should be equivalent to that expected by the function **strtoul** with a base argument of ten. See **strtoul** for more information. The corresponding argument should point to an **unsigned int**.

x Convert the token from hexadecimal notation to a signed integer. The format should be equivalent to that expected by the function **strtol** with a base argument of 16. See **strtol** for more information. The corresponding argument should point to an **unsigned int**.

X Same as **x**. In previous releases of **Let's C**, the modifier **X** meant that the corresponding argument pointed to a **long** instead of an **int**. This has been changed to conform to the ANSI Standard, and may require that some code be rewritten.

% Match a single percent sign '%'. Make no conversion or assignment.

[/] Scan a *scanset*, which is a set of characters enclosed by brackets. A character that matches any member of the scanset is copied into the area pointed to by the corresponding argument, which should be a **char** * that points to enough allocated memory to hold the

maximum number of characters that may be copied, plus the concluding null character. Appending a circumflex '^' to the scanset tells **scanf** to copy every character that does *not* match a member of the scanset (i.e., *complements* the scanset). If the format string begins with ']' or '^]', then ']' is included in the scanset, and the set specifier is terminated by the next ']' in the format string. If a hyphen appears within the scanset, the behavior is implementation-defined; often, it indicates a range of characters, as in **[a-z]**.

For example, passing the string **hello, world** to

```
char array[50];
scanf("[^abcd]", array);
```

writes the string **hello, worl** into **array**.

Cross-references

Standard, §4.9.6.4
The C Programming Language, ed. 2, p. 246

See Also

fscanf, printf, sscanf, STDIO

Notes

scanf will read up to, but not through, a newline character. The newline remains in the standard input device's buffer until you dispose of it. Programmers have been known to forget to empty the buffer before calling **scanf** a second time, which leads to unexpected results.

Experience has shown that **scanf** should not be used directly to obtain a string from the keyboard: use **gets** to obtain the string, and **sscanf** to format it.

The character that **scanf** recognizes as representing the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

scope — Definition

The term *scope* describes the portion of the program in which a given identifier is recognized, or *visible*. Scope is similar to, but not identical to, linkage. Linkage refers to whether an identifier can be joined, or *linked*, across files. Scope refers to the portion of a program that can recognize an identifier.

There are four varieties of scope: *block*, *file*, *function*, and *function prototype*.

An identifier with block scope is visible only within the block of code where it is declared. When the program reaches the '}' that ends that block of code, then the identifier is no longer visible, and so no longer "within scope".

An identifier with file scope is visible throughout the translation unit within which it is declared. The only identifiers that have file scope are those that are declared globally, i.e., that are declared outside the braces that enclose any function. If a function in one file uses an identifier that is defined in another file, it must mark that identifier as being external, by using the storage-class specifier **extern**.

An identifier with function scope is visible throughout a function, no matter where in the function it is declared. A label is the only variety of identifier that has function scope.

An identifier with function-prototype scope is visible only within the function prototype where it is declared. For example, consider the following function prototype:

```
void va_end(va_list listptr);
```

LEXICON

The identifier **listptr** has function-prototype scope. It is recognized only within that prototype, and is used only for purposes of documentation.

If an identifier is redeclared but is within an enclosing scope, it "hides" the outermost identifier and renders it inaccessible. This condition is called "information hiding", and it holds true as long as the inner declaration is within scope.

Example

The following program demonstrates scope, and shows how to hide information.

```
/* global i */
int i = 13;

void
function1(void)
{
      /* local i; hides global i */
      int i = 23;

      for(;;) {
            /* block-scope i; hides local and global i's */
            int i = 33;
            /* print block-scope i */
            printf ("block-scope i: %d\n", i);
            break;
      }
      /* block-scope i has disappeared; print local i */
      printf ("local i: %d\n", i);
}

void
function2(void)
{
      /* local i has disappeared; print global i */
      printf("global i: %d\n", i);
}

main(void)
{
      function1();
      function2();
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §3.1.2.1
The C Programming Language, ed. 2, p. 227

See Also

extern, identifiers, storage duration

Notes

If an identifier is declared both within a block and with the storage-class identifier **extern**, it has block scope. An external declaration made within one block of code is not available outside that block. If an identifier that is declared external within one block is referenced within another, behavior is undefined.

A common extension to C automatically promotes to file scope all external identifiers that are declared within a block. Under such implementations, the following will work correctly:

```
/* non-ANSI code! */
function1()
{
      extern float example();
         . . .
}

function2()
{
      float variable;
        . . .
      variable = example();
        . . .
}
```

Under the Standard, however, this code will not work correctly: the declaration of the function **example** has block scope; therefore, it cannot be seen in **function2**. In **function2**, therefore, the translator properly assumes that **example** returns an **int**. The **float** that **example** actually returns is altered, causing undefined behavior. ANSI C causes this code to behave differently than expected, and an implementation may not issue a warning message. This is a quiet change that may break existing code.

sequence point — Definition

A sequence point is any point in a program where all side effects are resolved. At every sequence point, the environment of the actual machine must match that of the abstract machine. That is, whatever optimizations or short-cuts an implementation may take, at every sequence point it must be as if the machine executed every instruction as it appeared literally in the program. Sequence points cause the program's actual behavior to be synchronized with the abstract behavior that the source code describes.

The sequence points are as follows:

- When all arguments to a function call have been evaluated.
- When the *first* operand of the following operators has been evaluated: logical AND '&&', logical OR '||', conditional '?', and comma ','.
- When a variable is initialized.
- When the controlling expression or expressions are evaluated for the following statements: **do**, **for**, **if**, **return**, **switch**, and **while**.

Cross-reference

Standard, §2.1.2.3

See Also

side effect, translation units

setbuf() — STDIO (libc)

Set alternative stream buffer
#include <stdio.h>
void setbuf(FILE **fp***, char** **buffer***);**

When the functions **fopen** and **freopen** open a stream, they automatically establish a buffer for it. The buffer is **BUFSIZ** bytes long. **BUFSIZ** is a macro that is defined in the header **stdio.h**.

setbuf changes the buffer for the stream pointed to by *fp* from its default buffer to *buffer*. It sets *buffer* to be **BUFSIZ** bytes long. To create a buffer of a size other than **BUFSIZ**, use **setvbuf**.

LEXICON

You should use **setbuf** after *fp* has been opened, but before any data have been read from or written to it.

If *buffer* is set to NULL, then *fp* will be unbuffered. For example, the call

```
setbuf(stdout, NULL);
```

ensures that all output to the standard output stream is unbuffered.

Cross-references

Standard, §4.9.5.5
The C Programming Language, ed. 2, p. 243

See Also

BUFSIZ, fclose, fflush, freopen, setbuf, setvbuf, STDIO

setjmp() — Non-local jump (setjmp.h)

Save environment for non-local jump
#include <setjmp.h>
int setjmp(jmp_buf *environment***);**

setjmp copies the current environment into the array **jump_buf**. The environment can then be restored by a call to the function **longjmp**.

environment is of type **jmp_buf**, which is defined in the header **setjmp.h**. **Let's C** defines **jmp_buf** to be an array of 11 **long**s.

setjmp returns zero if it is called directly. When it returns after a call to **longjmp**, however, it returns **longjmp**'s argument *rval*. If *rval* is set to zero, then **setjmp** returns one. See **longjmp** and **non-local jumps** for more information.

Cross-references

Standard, §4.6.1.1
The C Programming Language, ed. 2, p. 254

See Also

longjmp, jmp_buf, non-local jumps

Notes

Many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **setjmp** and **longjmp** will result in the creation of mysterious and irreproducible bugs. The use of **longjmp** to exit interrupt, exception, or signal handlers is particularly hazardous.

setjmp must be used as the controlling operand in a **switch** statement, as the controlling expression in an **if** statement, or as an operand in an equality expression. Any other use generates undefined behavior.

To conform with the Standard, **setjmp** is implemented as a macro.

setjmp.h — Header

Declarations for non-local jump
#include <setjmp.h>

setjmp.h is the header that contains declarations for the elements that perform a non-local jump. It contains the prototype for the function **longjmp**, and it defines the macro **setjmp** and the type **jmp_buf**.

Cross-references

Standard, §4.6
The C Programming Language, ed. 2, p. 254

See Also

header, jmp_buf, longjmp, non-local jump, setjmp

setlocale() — Localization (libc)

Set or query a program's locale
#include <locale.h>
char *setlocale(int *portion***, const char ****locale***);**

setlocale is a function that lets you set all or a portion of the locale information used by your program or query for information about the current locale.

portion is the portion of the locale that you wish to set or query. The Standard defines a number of manifest constants for this purpose, as follows:

LC_ALL
: Set or query all locale-specific information. Setting the locale affects all of the following locale categories.

LC_COLLATE
: Set or query information that affects collating functions. This affects the operation of the functions **strcoll** and **strxfrm**.

LC_CTYPE
: Set or query information about character handling. This affects he operation of all character-handling functions, except for **isdigit** and **isxdigit**. It also affects the operation of the functions that handle multibyte characters, i.e., **mblen**, **mbtowc**, **mbstowcs**, and **wcstombs**, **wctomb**.

LC_MONETARY
: Set or query all monetary-specific information as used in the structure **lconv**, which is initialized by the function **localeconv**.

LC_NUMERIC
: Set or query information for formatting numeric strings. This may change the decimal-point character used by string conversion functions and functions that perform formatted input and output. This may also affect the contents of the structure **lconv**.

LC_TIME
: Set or query information for formatting time strings. This changes the operation of the function **strftime**.

Setting *locale* to NULL tells **setlocale** that you wish to query information about the current locale rather than set a new locale.

setlocale returns a pointer to a string that contains the information needed to set or examine the locale. For example, the call

```
setlocale(LC_TIME, "");
```

returns a string that can be used to modify the time and date functions to conform to the requirements of the native locale. **setlocale** returns NULL if it does not recognize either *portion* or *locale*.

LEXICON

Cross-reference

Standard, §4.4.1.1

See Also

lconv, localeconv, localization

Notes

The Standard's section on compliance states that any program that uses locale-specific information does not strictly comply with the Standard. Therefore, any program that uses a locale other than the **C** locale *cannot* be assumed to be portable to every environment for which a conforming implementation of C has been written. *Caveat utilitor*.

setvbuf() — STDIO (libc)

Set alternative stream buffer
#include <stdio.h>
int setvbuf(FILE **fp***, char** **buffer***, int** *mode***, size_t** *size***);**

When the functions **fopen** and **freopen** open a stream, they automatically establish a buffer for it. The buffer is **BUFSIZ** bytes long. **BUFSIZ** is a macro that is defined in the header **stdio.h**.

setvbuf alters the buffer used with the stream pointed to by *fp* from its default buffer to *buffer*. Unlike the related function **setbuf**, it also allows you set the size of the new buffer as well as the form of buffering.

buffer is the address of the new buffer. *size* is its size, in bytes. *mode* is the manner in which you wish the stream to be buffered, as follows:

_IOFBF	Fully buffered
_IOLBF	Line-buffered
_IONBF	No buffering

These macros are defined in the header **stdio.h**. For more information on what these terms mean, see **buffering**.

You should call **setvbuf** after a stream has been opened but before any data have been written to or read from the stream. For example, the following give *fp* a 50-byte buffer that is line-buffered:

```
char buffer[50];
FILE *fp;

fopen(fp, "r");
setvbuf(fp, buffer, _IOLBF, sizeof(buffer));
```

On the other hand, the following turns off buffering for the standard output stream:

```
setvbuf(stdout, NULL, _IONBF, 0);
```

setvbuf returns zero if the new buffer could be established correctly. It returns a number other than zero if something went wrong or if an invalid parameter is given for *mode* or *size*.

Example

This example uses **setvbuf** to turn off buffering and echo.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
```

```
main(void)
{
      int c;

      if(setvbuf(stdin,  NULL, _IONBF, 0))
            fprintf(stderr, "Couldn't turn off stdin buffer\n");

      if(setvbuf(stdout,  NULL, _IONBF, 0))
            fprintf(stderr, "Couldn't turn off stdout buffer\n");

      while((c = getchar()) != EOF)
            putchar(c);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.5.6
The C Programming Language, ed. 2, p. 243

See Also

BUFSIZ, fclose, fflush, fopen, freopen, setbuf, STDIO

shellsort() — Extended function (libc)

Sort arrays in memory
void shellsort(char **data***, short** *n***, short** *size***, short (****comp***)());**

shellsort is a generalized algorithm for sorting arrays of data in primary memory. It uses D. L. Shell's sorting algorithm. **shellsort** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure.

Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result, depending on whether *p1* is less than, equal to, or greater than *p2*, respectively.

See Also

general utilities, qsort
The Art of Computer Programming, vol. 3, pp. 84*ff*, 114*ff*

Notes

shellsort differs from the sort function **qsort** in that it uses an iterative algorithm that does not require much stack.

short int — Type

A **short int** is a signed integral type. This type can be no smaller than a **char**, and no larger than an **int**.

A **short int** can encode any number between **SHRT_MIN** and **SHRT_MAX**. These are macros that are defined in the header **limits.h**. The former equals -32,767, and the latter +32,767.

The types **short**, **signed short**, and **signed short int** are all synonyms for **short int**.

LEXICON

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

int, long int, types

side effect — Definition

A *side effect* is any change to the execution environment that is caused by the program that accesses a volatile object, modifies an object, modifies a file, or calls a function that performs any of these tasks. An expression may generate side effects; a void expression exists just for the side effects it generates.

Cross-references

Standard, §2.1.2.3
The C Programming Language, ed. 2, p. 53

See Also

Environment, sequence point, translation phase

sig_atomic_t — Type

Type that can be updated despite signals

sig_atomic_t is an integral data type that is defined in the header **signal.h**. It defines the type of "atomic" object that can be accessed properly even if an asynchronous interrupt occurs.

Cross-reference

Standard, §4.7.1

See Also

signal handling, signal.h, volatile

Notes

When declaring objects of this type, you should use the type qualifier **volatile**; for example:

```
volatile sig_atomic_t save_state;
```

The **volatile** declaration tells the translator to re-read the object's value from memory each time it is used in an expression. When the program says to store the object, it should be stored immediately.

signal() — Signal handling (libc)

Set processing for a signal
#include <signal.h>
void (*signal(int *signame*, **void (***function*)**(int)))(int);**

signal is a function that tells the environment what to do when it detects a given interrupt, or "signal." *signame* names the signal to be handled, and *function* points to the signal handler (the function to be executed when *signame* is detected). *signame* may be generated by the environment itself (when it detects an error condition, for example), by the hardware (to indicate a bus error, timer event, or other hardware error condition), or by the program itself (by using the function **raise**).

If **signal** is successful, it returns a pointer to the function that the environment previously used to handle *signame*. If an error occurred, **signal** returns **SIG_ERR** and the global variable **errno** is set to an appropriate value. For a list of the signals recognized, see **signal handling**.

signal can establish the following ways of handling a *signame*:

1. If it sets *function* to **SIG_DFL**, it tells the environment to execute the default signal-handling function for *signame*.
2. Then, the equivalent of

   ```
   (*function)(signame)
   ```

   is executed, where function is the user-defined function installed with **signal** to handle *signame*.
3. If it sets *function* to point to a user-defined function, then it tells the environment to execute that function when it detects *signame*.

If **signal** is used to establish a user-defined *function* for a particular signal, then the following occurs when that signal is detected:

1. The equivalent of

   ```
   signal(signame, SIG_DFL);
   ```

   is executed. If *signame* is equivalent to **SIGILL** (which indicates that an illegal instruction has been found), then this step is optional, depending upon the implementation.
2. Then, the equivalent of

   ```
   (*function)(signame)
   ```

   is executed, where *function* points to a user-defined function. Some signals are reset to **STD_DFL**, some are not. The exception handler should be reset by another call to **signal** if subsequent signals are expected for that condition.
3. *function* can terminate either by returning to the calling function, or by calling **abort**, **exit**, or **longjmp**. If *function* returns and *signame* indicates that a computational exception had occurred (e.g., division by zero), then the behavior is undefined. Otherwise, the program which responded to the signal will continue to execute.

Cross-references

Standard, §4.7.1.1
The C Programming Language, ed. 2, p. 255

See Also

raise, signal handling, signal.h

Notes

The signal handler pointed to by *function* should not be another library function. Also, the signal handler should not attempt to modify external data other than those declared as type **volatile sig_atomic_t**.

signal.h — Header

Signal-handling routines
#include <signal.h>

signal.h is the header that defines or declares all elements used to handle asynchronous interrupts, or *signals*.

Signals vary from environment to environment. Therefore, the contents of **signal.h** will also vary greatly from environment to environment, and from implementation to implementation. The Standard mandates that it define the following elements to create a skeletal, portable suite of signal-

handling routines:

Type

sig_atomic_t	Type that can be accessed atomically
SIG_DFL	Default signal-handling indicator
SIG_ERR	Indicate error in setting a signal
SIG_IGN	Indicate ignore a signal
SIGABRT	Abort signal
SIGFPE	Erroneous arithmetic signal
SIGILL	Illegal instruction
SIGINT	Interrupt signal
SIGSEGV	Invalid access to storage signal
SIGTERM	Program termination signal

Functions

raise	Generate a signal
signal	Set processing for a signal

Cross-references

Standard, §4.7
The C Programming Language, ed. 2, p. 255

See Also

signal handling

signal handling — Overview

#include <signal.h>

A *signal* is an asynchronous interrupt in a program. Its use allows a program to be notified of and react to external conditions, such as errors that would otherwise force it either to abort or to continue despite erroneous conditions.

To respond to a signal, a program uses a *signal handler*, which is a function that performs the actions appropriate to a given signal. A signal handler usually is installed early in a program. It is invoked either when the condition arises for which the signal handler was installed, or when the program uses the function **raise** to raise a signal explicitly. A signal handler can be thought of as a "daemon," or a process that lives in the background and waits for the right conditions to occur for it to spring to life. Once the signal has been handled, the program may continue to execute.

Every conforming implementation of C must include at least a skeletal facility for handling signals. The Standard describes two functions: **raise**, which generates (or "raises") a signal; and **signal**, which tells the environment what function to execute in response to a given signal.

The suite of signals that can be handled varies from environment to environment. At a minimum, the following signals must be recognized:

SIGABRT	Abort
SIGFPE	Erroneous arithmetic
SIGILL	Illegal instruction
SIGINT	Interrupt
SIGSEGV	Invalid access to storage
SIGTERM	Program termination request

All of these are positive integral expressions. An implementation is obliged to respond only if one of these signals is raised explicitly via the function **raise**. This limitation is imposed because in some environments it may be impossible for an implementation to "sense" the presence of such conditions.

signal tells the environment which function to execute in response to a signal by passing it a pointer to that function. The Standard describes three macros that expand to constant expressions that point to functions, as follows:

SIG_DFL	Default signal-handling indicator
SIG_ERR	Indicate error in setting a signal
SIG_IGN	Indicate ignore a signal

The Standard describes a new data type, called **sig_atomic_t**. An object of this type can be updated or read correctly, even if a signal occurs while it is being updated or read. Accesses to objects of this type are atomic, i.e., uninterruptable.

All of the above are defined or declared in the header **signal.h**.

Cross-references

Standard, §4.7, §2.2.3
The C Programming Language, ed. 2, p. 255

See Also

Library, sequence points, signal.h, signals/interrupts

Notes

The name *signal* is derived from the electrical model of having a wire connected to the central processing unit for an interrupt. When the level on the wire would rise, an interrupt would be generated and the central processing unit would service the device that "raised" its "signal."

signals/interrupts — Definition

The Standard mandates the following restrictions upon the manner in which functions are implemented. First, a signal must be able to interrupt a function at any time. Second, a signal handler must be able to call a function without affecting the value of any object with automatic duration created by any earlier invocation of the function. Third, the function image (that is, the set of instructions that constitutes the executable image of the function) cannot be altered in any way as it is executed. All variables must be kept outside of the function image.

MS-DOS Interrupts

MS-DOS makes available to the programmer a series of interrupts that can be used to perform all manner of useful tasks. These interrupts and their functions can be accessed directly through the C function **intcall**.

The header **dos.h** defines a set of manifest constants that use most MS-DOS interrupts. The following table lists these constants, the interrupt and function number they define, and gives a brief description of what each does. Some constants combine two interrupts to form one function. For example, **CLRIN** combines interrupts 0x0C and 0x01.

Interrupt 10 (text mode)

GCDM	0x0F00	Get current display mode
IWDOWN	0x0700	Initialize window or scroll window down
IWUP	0x0600	Initialize window or scroll window up
RACCUR	0x0800	Read attribute & character at cursor
RDCP	0x0300	Read cursor positon
RGRPIX	0x0D00	Read graphics pixel
RLPP	0x0400	Read light pen position
SDP	0x0500	Select display page
SETCLR	0x0B00	Set color palette
SETCP	0x0200	Set cursor position

SETCT	0x0100	Set cursor type
SPALREG	0x1000	Set palette registers
WACCUR	0x0900	Write attribute and character at cursor
WCONLY	0x0A00	Write character only at cursor
WGRPIX	0x0C00	Write graphics pixel
WSTRING	0x1300	Write string (AT only)
WTELE	0x0E00	Write text in teletype mode

Interrupt 10 (graphics mode)

VM1620JR	0x0008	160x200 16-color graphics (*PCjr*)
VM3220C	0x0004	320x200 four-color graphics
VM3220CB	0x0005	320x200 four-color graphics color burst off
VM3220EG	0x000D	320x200 16-color graphics (EGA)
VM3220JR	0x0009	320x200 16-color graphics (*PCjr*)
VM4025BW	0x0000	40x25 black & white text, color ad.
VM4025C	0x0001	40x25 color text
VM64202	0x0006	640x200 two-color graphics
VM6420EG	0x000E	640x200 16-color graphics (EGA)
VM6420JR	0x000A	640x200 16-color graphics (*PCjr*)
VM64354E	0x0010	640x350 four- or 16-color graphics (EGA)RAM
VM6435EG	0x000F	640x350 monochrome graphics (EGA)
VM8025BW	0x0002	80x25 black & white text
VM8025C	0x0003	80x25 color text
VMMONOAD	0x0007	Monochome adapter text display

Interrupt 13

FORMDT	0x0500	Format disk track
GFDSS	0x0100	Get disk system status
RDFD	0x0200	Read disk
RSTFDS	0x0000	Reset disk system
VERDS	0x0400	Verify disk sectors
WRTDSK	0x0300	Write to disk

Interrupt 14

ITCOMP	0x0000	Initialize communications port
RCCOMP	0x0200	Read character from communications port
WCCOMP	0x0100	Write character to communications port

Interrupt 16

RCKEYB	0x0000	Read character from keyboard
RKEYST	0x0100	Read keyboard status
RTKEYF	0x0200	Return keyboard flags

Interrupt 17

INITPP	0x0100	Initialize printer port
PRNSRQ	0x0200	Request printer status
WCPRN	0x0000	Write character to printer port

Interrupt 21

ALLOC	0x4800	Allocate memory
BUFCON	0x0A00	Read console, buffered
CHDIR	0x3B00	Change current directory
CHMOD	0x4300	Change file mode

CLOSEF*	0x1000	Close a file
CLOSEH	0x3E00	Close a file
CLR_E	0x0C08	Clear console, accept input without echo
CLRBUF	0x0C0A	Clear console, accept buffered input
CLRIN	0x0C01	Clear console, echo console input
CLRDIO	0x0C06	Clear console, perform direct console I/O
CLRRAW	0x0C07	Clear console, accept raw input
CONSTAT	0x0B00	Return console/s status
CREATH	0x3C00	Create a file
CTLBCHK	0x3300	Get/set Ctrl-Break flag
DELETE	0x4100	Delete a file
DELETEF*	0x1300	Delete a file
DUPH	0x4500	Duplicate a file handle
EXEC	0x4B00	Load or execute a program
FDUPH	0x4600	Force a duplicate of handle
FFIRST*	0x1100	Search for first match
FNEXT*	0x1200	Search for next match
FREE	0x4900	Free allocated memory
GETALTI	0x1B00	Get allocation table information
GETCDI	0x3800	Get country-dependent information
GETCDIR	0x4700	Get current directory
GETDATE	0x2A00	Get date
GETDISK	0x1900	Get default disk drive
GETDTA	0x2F00	Get address of disk transfer area
GETFREE	0x3600	Get free disk space
GETTIME	0x2C00	Get time
GETVEC	0x3500	Get interrupt vector
GETVER	0x3000	Get MS-DOS version number
GETVST	0x5400	Get verify state
GSDT	0x5700	Get/set a file's date and time
IOCTLH	0x4400	I/O control for devices
LSEEKH	0x4200	Move file read/write pointer
MAKEF*	0x1600	Create or truncate a file
MKDIR	0x3900	Create a sub-directory
NEXIT	0x4C00	Terminate a process
NFFIRST	0x4E00	Search for first match
NFNEXT	0x4F00	Search for next match
OPENF*	0x0F00	Open a file
OPENH	0x3D00	Open a file
PROGSEG	0x2600	Create program segment
PUTSTR	0x0900	Output string, terminated with '$'
READB*	0x2700	Block read, random
READH	0x3F00	Read from a file or device
READR*	0x2100	Read, random
READS*	0x1400	Read sequential
RENAME	0x5600	Rename a file
RENAMEF*	0x1700	Rename a file
RESDSK	0x0D00	Reset disk system
RMDIR	0x3A00	Remove a sub-directory
SELDSK	0x0E00	Set default disk drive
SETBLK	0x4A00	Modify allocated memory blocks
SETDATE	0x2B00	Set date
SETDMAO	0x1A00	Set disk transfer address
SETINT	0x2500	Set interrupt vector

LEXICON

SETRREC*	0x2400	Set random record number
SETTIME	0x2D00	Set time
SIZEF*	0x2300	Compute size of file
TERMRES	0x3100	Terminate and remain resident
VERIFY	0x2E00	Disk write verification
WAIT	0x4D00	Get return code of subprocess
WRITEB*	0x2800	Block write, random
WRITEH	0x4000	Write to a file or device
WRITER*	0x2200	Write, random
WRITES*	0x1500	Write sequential

The interrupts marked with an asterisk '*' use the file control block. These functions, in general, have been replaced by other, similarly named functions that are easier to use. The file control block is a structure, defined as follows:

```
typedef struct fcb_t {
      unsigned char f_drive;          /* drive code (A=1, etc.) */
      char f_name[8],                 /* file name */
           f_ext[3];                  /* file suffix */
      unsigned short f_block;         /* current block
                                         (=128 records) */
      unsigned short f_recsz;         /* record size in bytes
                                         (=1) */
      unsigned long f_size;           /* file size, bytes
                                         (system) */
      unsigned int f_date;            /* modif. date (system) */
      char f_sys[10];                 /* for system use */
      unsigned char f_rec;            /* current record in block */
      unsigned long f_seek;           /* random record position */
} fcb_t;
```

Calling DOS Interrupts

Let's C offers two ways to use MS-DOS interrupts in your C programs.

The first is through the function **intcall**. **intcall** gives a convenient way to call an MS-DOS interrupt directly from a C program. For more information and examples on how to use this function, see the entry for **intcall**.

The other method is by using the programs **int.c** and **intdis.m**, whose source code is included with **Let's C**. Unlike **intcall**, which is a tool for calling MS-DOS interrupts, these programs allow i8086 interrupts to call you. Thus, they are a tool for building interrupt handlers. They also demonstrate how to combine a C program with one written in assembly language.

The suffix '.m' is unique to Mark Williams Company. It is used with a file of assembly language that is first treated by **cpp**, a command that invokes the C preprocessor. Thus, a '.m' file can contain conditionalized code, manifest constants, and all other commands that are recognized by the preprocessor. To compile such a file, assemble it through the **cc** command. For example, to assemble **foo.m**, use the command:

```
cc foo.m
```

cc will automatically call **cpp**, and pass its output to the assembler **as**. The entry for **as** presents an example of a **.m** program. See the entry on **larges.h** for more information on the **.m** format in general.

Example

The following example, called **example.c**, uses routines in **int.c** and **intdis.m** to call several MS-DOS interrupts. You should enter it into the directory where you have stored **int.c** and **intdis.m**, and compile it with the following command line:

```
cc example.c int.c intdis.m
```

This program works in both LARGE and SMALL model. Compile it with the command line

```
cc -VLARGE example.c int.c intdis.m
```

to create a LARGE-model executable.

```
#include <stdio.h>
#include <stdlib.h>

#define INT_BREAK 0x1B        /* keybd ctrl-break int */
#define INT_TICK 0x1C         /* system timer tick int */
#define STACKSIZE 0x100       /* small stack for locals */

int breakid;
int timerid;

#define TRUE 1
#define FALSE 0

int breakflag = FALSE;
int timerflag = FALSE;

/*
 *    Service routine for the Ctrl-Break Interrupt (0x1B).
 *    Simply sets the breakflag to TRUE.
 */

breaktrp(void)
{
      breakflag = TRUE;
      return(0);
}

/*
 *    Service routine for Timer-Tick Interrupt (0x1C).
 *    This comes from the 8253-5 Programmable Interval Timer
 *    at a rate of 18.2 Hz.  Thus every 91
 *    ( = 18.2 * 5 ) interrupts
 *    or 5 seconds, set the timerflag to TRUE.
 */

timertrp(void)
{
      static counter = 0;

      if(++counter == 91) {
            timerflag = TRUE;
            counter = 0;
      }

      /* Link in case interrupt 0x1C did something already */
      return(1);
}

void
fatal(char *message)
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}
```

LEXICON

```
main(void)
{
      int breaktrp();
      int timertrp();

      if ( (breakid=setint(INT_BREAK, breaktrp,
            STACKSIZE, 1)) == -1 )
            fatal("Error setting ctrl-break interrupt.");
      printf("Ctrl-Break Interrupt Set.\n");

      if((timerid=setint(INT_TICK, timertrp,
            STACKSIZE, 1)) == -1 )
            fatal("Error setting timer-tick interrupt.");
      printf("Timer-Tick Interrupt Set.0);

      for (;;) {
            if(breakflag == TRUE)
                  break;
            if(timerflag == FALSE)
                  continue;
            printf("Another 5 sec gone.\n");
            timerflag = FALSE;
      }
      printf("Got the Ctrl-Break Key.\n");

      if(clearint(breakid) != 0)
            fatal("Unable to reset interrupt.");
      printf("Ctrl-Break interrupt reset.\n");

      if ( clearint(timerid) != 0 ) {
            fatal("Unable to reset Timer-Tick Interrupt.");
      printf("Timer-Tick interrupt reset.\n");

      return EXIT_SUCCESS;
}
```

Cross-references

Standard, §2.2.3 *Advanced MS-DOS*, pp 208*ff*, 272*ff*

See Also

Environment, signal handling

signed — Definition

The modifier **signed** indicates that a data type can contain both positive and negative values. In some representations, the sign of a signed object is indicated by a bit set aside for the purpose. For this reason, a signed object can encode an absolute value only half that of its unsigned counterpart.

The four integral data types can be marked as signed: **char**, **short int**, **int**, and **long int**.

The implementation defines whether a **char** is signed or unsigned by default. The Standard describes the types **signed char** and **unsigned char**. These let the programmer use the type of **char** other than that supplied by the implementation. **short int**, **int**, and **long int** are signed by default. The declarations **signed short int**, **signed int**, and **signed long int** were created for the sake of symmetry.

For information about converting one type of integer to another, see **integral types**.

If **signed** is used by itself, it is a synonym for **int**.

Cross-references

Standard, §3.1.2.5, §3.2.1.2
The C Programming Language, ed. 2, p. 211

See Also

types, unsigned

signed char — Type

A **signed char** is a type that has the same size and the same alignment requirements as a plain **char**. The Standard created this type for implementations whose **char** type is unsigned by default.

A **signed char** can encode values from **SCHAR_MIN** to **SCHAR_MAX**. These are macros that are defined in the header **limits.h**. The former is set to -127, and the latter to +127.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.5.2
The C Programming Language, ed. 2, p. 44

See Also

char, types, unsigned char

sin() — Mathematics (libm)

Calculate sine
#include <math.h>
double sin(double *radian***);**

sin calculates and returns the sine of its argument *radian*, which must be in radian measure.

Example

This example verifies the identity **sin(2*x) == 2*sin(x)*cos(x)** over a range of values. Then, it scans the range of the worst error in smaller and smaller increments, until the precision of the floating point will not allow any more.

```
#include <float.h>
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define PI 0.31415926535897932e+01

main(void)
{
      int ct;
      double a, e, i, worstp;
      double worste=0.0;
      double f=-PI;
```

```
      printf("Verify sin(2*x) == 2*sin(x)*cos(x)\n");
      for(i = (PI / 100.0); (f + i) > f; i *= 0.01) {
            for(ct = 200, a = f; --ct; a += i) {
                  e = fabs(sin(a+a)-(2.0*sin(a)*cos(a)));
                  if(e > worste) {
                        worste = e;
                        worstp = a;
                  }
            }
            f = worstp - i;
      }

      printf("Worst error %.17e at %.17e\n", worste, worstp);
      printf("sin(2x)=%.17e 2*sin(x)*cos(x)=%.17e\n",
            f=sin(worstp+worstp), 2.0*sin(worstp)*cos(worstp));
      printf("Epsilon is %.17e\n", fabs(f) * DBL_EPSILON);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.5.2.6
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan, atan2, cos, mathematics, tan

sinh() — Mathematics (libm)

Calculate hyperbolic sine
#include <math.h>
double sinh(double *value***);**

sinh calculates and returns the hyperbolic sine of *value*. A range error will occur if the argument is too large.

Cross-references

Standard, §4.5.3.2
The C Programming Language, ed. 2, p. 251

See Also

cosh, mathematics, tanh

size — Command

Print the size of an object module
size *file…*

size prints the size of each segment of each given *file*, which must be a relocatable object module or an executable file. The total size is given in decimal, and the size of each segment is given in both decimal and hexadecimal. All sizes are in bytes.

When it is used to size an executable file, **size** prints the size of the code segment and the data segment separately (in LARGE model), or the code segment plus the data segment (in SMALL model). Thus, **size** can help you to tell a SMALL-model program from one in LARGE model.

See Also

cc, commands, cpp, nm, strip

sizeof — C keyword

The operator **sizeof** yields the size of its argument, in bytes. Its argument can be the name of a type, an array, a function, a structure, or an expression that yields an object.

When the name of a type is used as the operand to **sizeof**, it must be enclosed within parentheses. If any of the types **char**, **signed char**, or **unsigned char** are used as the argument to **sizeof**, the result by definition is always one. When any complete type is used (i.e., a type whose size is known by the translator), the result is the size of that type, in bytes. For example,

```
sizeof (long double);
```

returns the size of a **long double** in bytes.

If **sizeof** is given the name of an array, it returns the size of the array. For example, the code

```
int example[5];
   . . . /* example[] is filled with some things */
sizeof example[] / sizeof int;
```

yields the number of members in **example[]**.

When **sizeof** is given the name of a structure or a **union**, it returns the size of that object, including padding used to align the objects within the structure, if any. This is especially useful when allocating memory for a linked list; for example:

```
struct example {
      int member1;
      example *member2;
};
struct example *variable;
variable=(struct example *)malloc(sizeof(struct example));
```

If **sizeof** is used to measure either a function or an array that has been passed as an argument to a function, it returns the size of a *pointer* to the appropriate object. This is because when an array name or function name is passed as an argument to a function, it is converted to a pointer. See **function definition** for more information.

sizeof always returns an object of type **size_t**; this type is defined in the header **stddef.h**. It is intended to be an unsigned integral type.

sizeof must not be used with a function, with an object whose type is incomplete, or a bit-field.

Example

For an example of using this operator in a program, see **bsearch**.

Cross-references

Standard, §3.3.3.4
The C Programming Language, ed. 2, p. 204

See Also

expressions, operators, size_t

SMALL model — Technical information

Intel single-segment memory model

The i8086/88 microprocessor uses a *segmented architecture*. This means that the memory is divided into segments of 64 kilobytes each; no program or data element can exceed that limit.

Intel Corporation has devised a number of memory models for handling segmented memory.

Let's C implements the two most useful of these: SMALL model and LARGE model.

SMALL model C programs use 16-bit pointers and NEAR calls. Because a 16-bit pointer can address 65,536 bytes (64 kilobytes) of memory, SMALL model programs are limited to 64 kilobytes (one segment) of code and 64 kilobytes of data.

The SMALL-model pointer consists only of the *offset* within a given segment, and does not include the *segment* itself. If you use a function that requires the full offset/segment pair, e.g., **_copy**, **peek**, or **poke**, you can supply the missing segment either by reading the contents of the DS segment register with the function **dsreg**, or by using the macro **PTR**. See the entries for **dsreg** and **PTR** for more information.

Note, too, that the SMALL-pointer is the same length as an **int**. This allows a programmer to use these data types interchangably. Most often, this happens when a programmer fails to declare properly a function that returns a pointer, so that the function is implicitly declared by the compiler as returning an **int**. Programs with this error will run correctly when compiled into SMALL model, but will fail to work when compiled into LARGE model. See the entry on **pun** for more information.

When **Let's C** compiles a program with the **-VSMALL** option, the resulting object module follows the rules of the *SMALL model*. This is the default setting for the compiler.

See Also

i8086 support, LARGE model, model, pun, technical information

source file — Definition

A *source file* is any file of C source text.

Cross-reference

Standard, §2.1.1.1

See Also

Environment, translation unit

sprintf() — STDIO (libc)

Print formatted text into a string
#include <stdio.h>
int sprintf(char **string***, const char** **format***, ...);**

sprintf constructs a formatted string in the area pointed to by *string*, and appends a null character onto the end of what it constructs. It translates integers, floating-point numbers, and strings into a variety of text formats.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how to convert a particular data type into text. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence "%%".) See **printf** for further discussion of the conversion specification, and for a table of the type specifiers that can be used with **sprintf**.

After *format* can come one or more arguments. There should be one argument for each conversion specification in *format*. The argument should be of the type appropriate to the conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *format* should be followed by three arguments, respectively, an **int**, a **long**, and a **char ***.

If there are fewer arguments than conversion specifications, then **sprintf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding conversion specifier,

then the behavior of **sprintf** is undefined. Thus, presenting an **int** where **sprintf** expects a **char** * may generate unwelcome results.

sprintf returns the number of characters written into *string*, not counting the terminating null character.

Cross-references

Standard, §4.9.6.5
The C Programming Language, ed. 2, p. 245

See Also

fprintf, printf, STDIO, vfprintf, vprintf, vsprintf

Notes

string must point to enough allocated memory to hold the string **sprintf** constructs, or you may overwrite unallocated memory.

The character that **sprintf** uses to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

Because the **printf** routines that print floating-point numbers are quite large, they are included only optionally. If you wish to have **printf** print **float**s or **double**s, you must compile your program with the **-f** option to the **cc** command. See **cc** for more details.

sqrt() — Mathematics (libm)

Calculate the square root of a number
#include <math.h>
double sqrt(double *z***);**

sqrt calculates and returns the square root of *z*.

Example

This example calculates the time an object takes to fall to the ground at sea level. It ignores air friction and the inverse square law.

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

double
fallingTime(double meters)
{
      double time;

      errno = 0;
      time = sqrt(meters * 2 / 9.8);
      /*
       * it would be simpler to test for (meters < 0) first,
       * but this way shows how sqrt() sets errno
       */
      if(errno) {
            printf("Sorry, but you can't fall up\n");
            return(HUGE_VAL);
      }
      return(time);
}
```

```
main(void)
{
      for(;;) {
            char buf[80];
            double height;

            printf("Enter height in meters ");
            fflush(stdout);
            if(gets(buf) == NULL || !strcmp(buf, "quit"))
                  break;

            errno = 0;
            height = strtod(buf, (char **)NULL);

            if(errno) {
                  printf("%s: invalid floating-point number\n");
                  continue;
            }

            printf("It takes %3.2f sec. to fall %3.2f meters\n",
                  fallingTime(height), height);

      }

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.5.5.2
The C Programming Language, ed. 2, p. 251

See Also

domain error, mathematics, pow

Notes

If *z* is negative, a domain error occurs.

srand() — General utility (libc)

Seed pseudo-random number generator
#include <stdlib>
void srand(unsigned int *seed***);**

srand uses *seed* to initialize the sequence of pseudo-random numbers returned by **rand**. Different values of *seed* produce different sequences.

Example

This example uses the random-number generator to encrypt or decrypt a file. This example is for illustration only. Do *not* use it if any serious attack is expected. This example also demonstrates a simple form of hashing.

```
#include <stdio.h>
#include <stdlib.h>

/* Ask for a string and echo it. */
char *
ask(char *msg)
{
      static char reply[80];
```

```
      printf("Enter %s ", msg);
      fflush(stdout);

      if(gets(reply) == NULL)
            exit(EXIT_SUCCESS);
      return(reply);
}

main(void)
{
      register char *kp;
      register int c, seed;
      FILE *ifp, *ofp;

      if((ifp = fopen(ask("input filename"), "rb")) == NULL)
            exit(EXIT_FAILURE);
      if((ofp = fopen(ask("output filename"), "wb")) == NULL)
            exit(EXIT_FAILURE);

      /* hash encryption key into an int */
      seed = 0;
      for(kp = ask("encryption key"); c = *kp++; ) {
            /* don't lose any bits */
            if((seed <<= 1) < 0)
                  /* a number picked at random */
                  seed ^= 0xE51B;
            seed ^= c;
      }

      /* initialize random-number stream */
      srand(seed);

      while((c = fgetc(ifp)) != EOF)
            /*
             * Use only the high byte of rand;
             * its low-order bits are very non-random
             */
            fputc(c ^ (rand() >> 8), ofp);

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.2.2
The C Programming Language, ed. 2, p. 252

See Also

general utilities, rand

sscanf() — STDIO (libc)

Read and interpret text from a string
#include <stdio.h>
int sscanf(const char **string***, const char** **format***, ...);**

sscanf reads characters from *string* and uses the string pointed to by *format* to interpret what it has read into the appropriate type of data. *format* points to a string that contains one or more conversion specifications, each of which is introduced with the percent sign '%'. For a table of the conversion specifiers that can be used with **sscanf**, see **scanf**.

After *format* can come one or more arguments. There should be one argument for each conversion specification in *format*, and the argument should point to a data element of the type appropriate to

the conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *format* should be followed by three arguments, pointing, respectively, to an **int**, a **long**, and an array of **char**s.

If there are fewer arguments than conversion specifications, then **sscanf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding type specification, then **sscanf** returns.

sscanf returns the number of input elements it scanned and formatted. If an error occurs while **sscanf** is reading its input, it returns **EOF**.

Example

This example reads a list of hexadecimal numbers from the standard input and adds them.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main(void)
{
      long h[5], total;
      char buf[80];
      int  count, i;

      printf("Enter a list of up to five hex numbers or quit\n");
      while(gets(buf) != NULL) {
            if(!strcmp("quit", buf))
                  break;

            count = sscanf(buf, "%lx %lx %lx %lx %lx",
                  h, h+1, h+2, h+3, h+4);

            for(i = total = 0; i < count; i++)
                  total += h[i];
            printf("Total 0x%lx %ld\n", total, total);
      }

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.6.6
The C Programming Language, ed. 2, p. 246

See Also

fscanf, printf, STDIO, scanf

Notes

sscanf is best used to read data you are certain are in the correct format, such as data previously written with **sprintf**.

The character that **sscanf** recognizes as representing the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

stack — Definition

The **stack** is the segment of memory that holds function arguments, local variables, function return addresses, and stack frame linkage information.

If your program uses recursive algorithms, or declares large amounts of automatic data, or simply contains many levels of functions calls, the stack may "overflow", and overwrite the program data.

By default, **Let's C** sets the default stack size to 2,048 bytes (two kilobytes). To increase the amount of stack available to your program, use the **-ys** option to the **cc** command. For example, to give the program **foo.c** 10,000 bytes of stack, use the following **cc** command:

```
cc -ys10000 foo.c
```

See Also

cc, Definitions

Standard — Overview

The *Standard* is the document written by the American National Standards Institute committee X3J11 to describe the programming language C. It is based on the following documents:

- Kernighan, B. W., Ritchie, D. M.: *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall Inc., 1978. The Standard bases its description of C syntax upon Appendix A of this book.
- /usr/group Standard Committee: *1984 /usr/group Standard*. Santa Clara, Calif.: /usr/group, 1984. This document was the basis for the Standard's description of the C library.
- *American National Dictionary for Information Processing Systems*. Information Processing Systems Technical Report ANSI X3/TR-1-82. 1982.
- ISO 646-1983 Invariant Code Set. This was used to help describe the C character set, and to select the characters that need to be represented by trigraphs.
- *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754-1985. This is the basis for the Standard's description of floating-point numbers.
- ISO 4217 Codes for Representation of Currency and Funds. This is the target for the Standard's description of locale-specific ways to represent money.

The first two, due to their fundamental effect upon the Standard, are referred to as the "base documents".

Cross-reference

Standard, §1.3, §1.5

See Also

Definitions, Environment, Language, Library, DOS-specific features

standard error — Definition

When a C program begins, it opens three text streams by default: the standard error, the standard input, and the standard output. The *standard error* is the stream into which error messages are written. In most implementations, the standard error stream is associated with the user's terminal.

The macro **stderr** points to the **FILE** object through which the standard error device is accessed. It is defined in the header **stdio.h**.

LEXICON

Cross-references

Standard, §4.9.3
The C Programming Language, ed. 2, pp. 151*ff*

See Also

standard input, standard output, stderr, STDIO

standard input — Definition

When a C program begins execution, it opens three text streams by default: the standard error, the standard input, and the standard output. The *standard input* is the stream from which the program receives input by default. In most implementations, the standard input stream is associated with the user's terminal.

The macro **stdin** points to the **FILE** object that accesses the standard input stream. It is defined in the header **stdio.h**.

Cross-references

Standard, §4.9.3
The C Programming Language, ed. 2, pp. 151*ff*

See Also

standard error, standard output, stdin, STDIO

standard output — Definition

When a C program begins execution, it opens three text streams by default: the standard output, the standard input, and the standard error. The *standard output* is the stream into which a program's non-diagnostic output is written. In most implementations, the standard output stream is associated with the user's terminal.

The macro **stdout** points to the **FILE** object that accesses the standard output device. It is defined in the header **stdio.h**.

Cross-references

Standard, §4.9.3
The C Programming Language, ed. 2, pp. 151*ff*

See Also

standard error, standard input, STDIO, stdout

stat() — Access checking (libc)

Find file attributes
#include <stat.h>
short stat(char **file***, struct stat** **statptr***);**

stat returns a structure that contains the attributes of a file. This function is included to maintain compatibility with the UNIX and COHERENT operating systems.

file points to the path name of file, and *statptr* points to a structure of the type **stat**, as defined in the header file **stat.h**.

The following summarizes the structure **stat**:

```
struct stat {
      unsigned short st_mode;            /* mode */
      long st_size;                      /* size, in bytes */
      struct dostime st_dostime;         /* MS-DOS time and date */
      time_t st_mtime;                   /* modification time */
};
```

The structure **dostime** is defined in the header file **dosfind.h**. The following lists the legal values for **st_mode**, which sets the file's attributes:

S_IFMT	0x0300	type
S_IFDIR	0x0100	directory
S_IFREG	0x0200	regular file
S_IREAD	0x0400	read permission; always 1
S_IWRITE		0x0800write permission

The entry **st_size** gives the size of the file, in bytes.

stat returns -1 if an error occurs, e.g., the file cannot be found. Otherwise, it returns zero.

Example

The following example, called **test.c**, demonstrates **stat**. When compiled, it will take a file name as an argument; it will then search for the file and, if it is found, print a summary of its status.

```
#include <stat.h>
#include <stdio.h>
#include <stdlib.h>
char *_cmdname = "TEST";

void
fatal(char *error)
{
      fprintf(stderr, "Fatal Error: %s\n", error);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      char *name;
      struct stat status;

      if ( argc != 2 )
            fatal("Usage: command filename");
      name = argv[1];
      if (stat(name, &status) != 0)
            fatal("Can't find file");

      printf("File: {%s}\n", name);
      printf("st_mode: 0x%x\n", status.st_mode);
      printf("st_size: %D\n", status.st_size);
      printf("st_dostime: %02d-%02d-%02d %02d:%02d:%02d\n",
            status.st_dostime.dos_month,
            status.st_dostime.dos_day,
            status.st_dostime.dos_year+80,
            status.st_dostime.dos_hour,
            status.st_dostime.dos_minute,
            status.st_dostime.dos_twosec*2);
      printf("st_mtime: %s", ctime(&status.st_mtime));
      return EXIT_SUCCESS;
}
```

LEXICON

See Also

access checking, open, stat.h

stat.h — Header

Definitions and declarations to obtain file status
#include <stat.h>

stat.h is a header file that contains the declarations of several structures used by the routine **stat**, which returns information about a file's status.

See Also

access checking, header, stat

statements — Overview

A *statement* specifies an action to be performed. Unless otherwise specified, statements are executed in the order in which they appear in the program.

The actions of some statements may be controlled by a *full expression*; this is an expression that is not part of another expression. For example, **do**, **if**, **for**, **switch**, and **while** introduce statements that are controlled by one or more full expressions. The **return** statement may also use a full expression.

The Standard describes the following varieties of statements:

Compound statement

Expression statement

Iteration statements
do
for
while

Jump statements
break
continue
goto
return

Labelled statements
case
default

Null statement

Selection statements
if
else
switch

The set of compound, iteration, and selection statements is the foundation upon which many programming languages are based. From these alone, a programmer can construct many useful and interesting programs.

Let's C also includes the keyword **alien**, which marks a function that uses non-C calling conventions.

Cross-references

Standard, §3.6
The C Programming Language, ed. 2, pp. 222*ff*

See Also

alien, Language

static — C keyword

Internal linkage
static *type identifier*

The storage-class specifier **static** declares that *identifier* has internal linkage. This specifier may not be used to declare a function that has block scope.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, p. 83

See Also

linkage, storage-class identifiers

stdarg.h — Header

Header for variable numbers of arguments
#include <stdarg.h>

The header **stdarg.h** declares and defines routines that are used to traverse a variable-length argument list. It declares the type **va_list** and the function **va_end**, and it defines the macros **va_start** and **va_arg**.

Cross-references

Standard, §4.8
The C Programming Language, ed. 2, p. 254

See Also

header, variable arguments

stderr — Macro

Pointer to standard error stream
#include <stdio.h>

When a C program begins, it opens three text streams by default: the standard error, the standard input, and the standard output. **stderr** points to the **FILE** object through which the standard error stream is accessed; this is the stream into which error messages are written. In most implementations, the standard error stream is associated with the user's terminal.

stderr is defined in the header **stdio.h**.

stderr is not fully buffered when it is opened.

Example

For an example of **stderr** in a program, see **fprintf**.

Cross-references

Standard, §4.9.1, §4.9.3
The C Programming Language, ed. 2, p. 243

See Also

stdin, stdout, standard error, STDIO, stdio.h

LEXICON

stdin — Macro

Pointer to standard input stream
#include <stdio.h>

When a C program begins, it opens three text streams by default: the standard error, the standard input, and the standard output. **stdin** points to the **FILE** object that accesses the standard input stream; this is the stream from which the program receives input by default. In most implementations, the standard input stream is associated with the user's terminal.

stdin is defined in the header **stdio.h**.

Example

For an example of **stdin** in a program, see **setvbuf**.

Cross-references

Standard, §4.9.1, §4.9.3
The C Programming Language, ed. 2, p. 243

See Also

stderr, stdout, standard input, STDIO, stdio.h

STDIO — Overview

Standard input and output
#include <stdio.h>

STDIO is an acronym for *standard input and output*. Input-output can be performed on text files, binary files, or interactive devices. It can be either buffered or unbuffered.

The Standard describes 41 functions that perform input and output, as follows:

Error handling

clearerr	Clear a stream's error indicator
feof	Examine a stream's end-of-file indicator
ferror	Examine a stream's error indicator
perror	Write error message into standard error stream

File access

fclose	Close a stream
fflush	Flush an output stream's buffer
fopen	Open a stream
freopen	Close and reopen a stream
setbuf	Set an alternate buffer for a stream
setvbuf	Set an alternate buffer for a stream

File operations

remove	Remove a file
rename	Rename a file
tmpfile	Create a temporary file
tmpnam	Generate a unique name for a temporary file

File positioning

fgetpos	Get value of stream's file-position indicator (**fpos_t**)
fseek	Set stream's file-position indicator
fsetpos	Set stream's file-position indicator (**fpos_t**)
ftell	Get the value of the file-position indicator
rewind	Reset stream's file-position indicator

Input-output

By character

fgetc	Read a character from a stream
fgets	Read a line from a stream
fputc	Write a character into a stream
fputs	Write a string into a stream
getc	Read a character from a stream
getchar	Read a character from the standard input stream
gets	Read a string from the standard input stream
putc	Write character into a stream
putchar	Write a character into the standard output
puts	Write a string into the standard output
ungetc	Push a character back into the input stream

Direct

fread	Read data from a stream
fwrite	Write data into a stream

Formatted

fprintf	Print formatted text into a stream
fscanf	Read formatted text from a stream
printf	Format and print text into standard output stream
scanf	Read formatted text from standard input stream
sprintf	Print formatted text into a string
sscanf	Read formatted text from string
vfprintf	Format and print text into a stream
vprintf	Format and print text into standard output stream
vsprintf	Format and print text into a string

The prototypes for these functions appear in the header **stdio.h**, along with definitions for the types and macros they use.

All STDIO functions access a file or device through a *stream*. A stream is accessed via an object of type **FILE**; this object contains all of the information needed to access the file or device under the given environment. Because of the heterogeneous environments under which C has been implemented, the Standard does not describe the interior workings of the **FILE** object. It states only that this object contain all information needed to access a stream under the given environment.

Cross-references

Standard, §4.9
The C Programming Language, ed. 2, pp. 151*ff*, 241*ff*

See Also

close, create, extended STDIO, file, file-position indicator, Library, line, open, stdio.h, stream

Notes

Let's C also includes the following extended functions and macros that perform STDIO tasks:

_exit	Exit from a program without clean-up
close	Close a file
creat	Create a file
dup	Duplicate a file descriptor
dup2	Duplicate a file descriptor
execall	Pass arguments to a program
fdopen	Use a file descriptor to open a stream
fgetw	Read a word from a stream

LEXICON

fileno	Get a file descriptor
fputw	Write a word into a stream
getanb	Read unbuffered from auxiliary port
getcnb	Read unbuffered from the console
getw	Read a word from a stream
in	Read a word from a port
inb	Read a byte from a port
lseek	Set stream's file-position indicator
open	Open a file
out	Write a word to a port
outb	Write a byte to a port
putanb	Write unbuffered to auxiliary port
putcnb	Write unbuffered to the console
putw	Write a word into a stream
read	Read data from a stream
regtop	Convert register pair to pointer
tempnam	Generate a unique name for a temporary file
unlink	Remove a file
write	Write data into a stream

The ANSI Standard forbids any ANSI header to declare or define any function or macro that is not described within the Standard. Therefore, the routines **fdopen**, **fgetw**, **fileno**, **fputw**, **getanb**, **getcnb**, **getw**, **putanb**, **putcnb**, **putw**, and **regtop** have been moved from header **stdio.h** into a new header, **xstdio.h**.

Any programs that uses any of these extended functions will not comply strictly with the Standard, and may not be portable to other compilers or environments.

stdio.h — Header

Declarations and definitions for STDIO

stdio.h is the header that holds the definitions, declarations, and function prototypes used by the STDIO routines.

The following lists the types, macros, and manifest constants defined in **stdio.h**:

Types

FILE	Hold descriptor for a stream
fpos_t	Hold current position within a file

Cross-references

Standard, §4.9.1
The C Programming Language, ed. 2, pp. 151*ff*, 241*ff*

See Also

header, STDIO

stdlib.h — Header

General utilities
#include <stdlib.h>

stdlib.h is a header that declares the Standard's set of general utilities and defines attending macros and data types, as follows:

Types

div_t	Type of object returned by **div**
ldiv_t	Type of object returned by **ldiv**

EXIT_FAILURE	Value to indicate that program failed to execute properly
EXIT_SUCCESS	Value to indicate that program executed properly
MB_CUR_MAX	Largest size of multibyte character in current locale
MB_LEN_MAX	Largest overall size of multibyte character in any locale
RAND_MAX	Largest size of pseudo-random number

Functions

abort	End program immediately
abs	Compute the absolute value of an integer
atexit	Register a function to be executed at exit
atof	Convert string to floating-point number
atoi	Convert string to integer
atol	Convert string to long integer
bsearch	Search an array
calloc	Allocate dynamic memory
div	Perform integer division
exit	Terminate a program gracefully
free	De-allocate dynamic memory to free memory pool
getenv	Read environmental variable
labs	Compute the absolute value of a long integer
ldiv	Perform long integer division
malloc	Allocate dynamic memory
mblen	Compute length of a multibyte character
mbstowcs	Convert multibyte-character sequence to wide characters
mbtowc	Convert multibyte character to wide character
qsort	Sort an array
rand	Generate pseudo-random numbers
realloc	Reallocate dynamic memory
strtod	Convert string to floating-point number
strtol	Convert string to long integer
strtoul	Convert string to unsigned long integer
system	Suspend a program and execute another
wcstombs	Convert wide-character sequence to multibyte characters
wctomb	Convert wide character to multibyte character

Cross-references

Standard, §4.10.1
The C Programming Language, ed. 2, p. 251

See Also

general utilities

LEXICON

stdout — Macro

Pointer to standard output stream
#include <stdio.h>

When a C program begins, it opens three text streams by default: the standard error, the standard input, and the standard output. **stdout** points to the **FILE** object that accesses the standard output stream. This is the stream into which non-diagnostic output is written. Under **Let's C**, the standard output stream is associated with the user's terminal.

stdout is defined in the header **stdio.h**.

Example

For an example of **stdout** in a program, see **setvbuf**.

Cross-references

Standard, §4.9.1, §4.9.3
The C Programming Language, ed. 2, p. 243

See Also

stdin, stderr, standard output, STDIO, stdio.h

stime() — Extended function (libc)

Set the operating system time
#include <time.h>
#include <xtime.h>
int stime(time_t *_timep_);

stime sets the operating system time, which **Let's C** defines as being the number of seconds since midnight of January 1, 1970, 0h00m00s UTC. The argument *timep* points to the new system time, which is of the type **time_t**. This is defined in the header file **time.h** as being equivalent to a **long**.

stime returns -1 on error, zero otherwise.

Example

The following example prints the time, then uses **stime** to reset the time by one hour.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void)
{
      time_t tp;

      /* print current time */
      time(&tp);
      printf("%s\n", ctime(&tp));

      /* subtract one hour (3600 seconds) from current time */
      tp -= 3600;
      if (stime(&tp) == -1) {
            printf("Cannot reset time.\n");
            exit(EXIT_FAILURE);
      }

      /* print altered time */
      time(&tp);
      printf("%s\n", ctime(&tp));
```

```
      /* add one hour to current time, to correct above */
      tp += 3600;
      if (stime(&tp) == -1) {
            printf("Cannot re-reset time.\n");
            exit(EXIT_FAILURE);
      }
      /* print fixed time, to confirm correction */
      time(&tp);
      printf("%s\n", ctime(&tp));
      return EXIT_SUCCESS;
}
```

See Also

extended time

Notes

To conform with the ANSI Standard, this function has been moved from the header **time.h** to the header **xtime.h**. This may require that some code be altered.

storage-class specifiers — Overview

A *storage-class specifier* specifies the manner in which an object is to be stored in memory. There are five such specifiers:

auto	Automatic storage duration
extern	External linkage
register	Quick access required
static	Internal linkage
typedef	Synonym for another type

Only one storage-class specifier is allowed per declaration. The Standard declares as "obsolescent" any declaration that does not have its storage class as the first specifier in a declaration.

Strictly speaking, **typedef** is not a storage-class specifier. The Standard bundles it into this group for the sake of convenience.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, p. 210

See Also

declarations, storage class, storage duration

storage duration — Definition

The term *storage duration* refers to how long a given object is retained within memory. There are two varieties of storage duration: *static* and *automatic*.

An object with static storage duration is retained throughout program execution. Its storage is reserved, and the object is initialized only when the program begins execution. All string literals have static duration, as do all objects that are declared globally — that is, declared outside of any function.

An object with automatic duration is declared within a block of code. It endures within memory only for the life of that block of code. Memory is allocated for the variable whenever that block is entered and deallocated when the block is terminated, either by encountering the '}' that closes the block, or by exiting the block with **goto**, **longjmp**, or **return**.

LEXICON

A common practice is to declare all automatic variables at the beginning of a function. These variables endure as long as the function is operating. If the function calls another function, then these functions are stored away (usually in an special area of memory called the "stack"), but they cannot be accessed until the called function returns.

Cross-references

Standard, §3.1.2.4
The C Programming Language, ed. 2, p. 195

See Also

auto, identifiers, scope, static

strcat() — String handling (libc)

Append one string onto another
char *__**strcat(char** *__*string1*__**, const char** *__*string2*__**);**

strcat copies all characters in *string2*, including the terminating null character, onto the end of the string pointed to by *string1*. The null character at the end of *string1* is overwritten by the first character of *string2*.

strcat returns the pointer *string1*.

Example

The following example concatenates two strings.

```
#include <stdio.h>
#include <string.h>

char string1[80] = "The first string. ";
char string2[] = "The second string.";

main(void)
{
      printf("result = %s\n", strcat(string1, string2));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.3.1
The C Programming Language, ed. 2, p. 250

See Also

string handling, strncat

Notes

string1 should point to enough reserved memory to hold itself and *string2*. Otherwise, data or code will be overwritten.

strchr() — String handling (libc)

Find a character in a string
#include <string.h>
char *__**strchr(const char** *__*string*__**, int** *__*character*__**);**

strchr searches for *character* within *string*. The null character at the end of *string* is included within the search. It is equivalent to the non-ANSI function **index**.

Internally, **strchr** converts *character* from an **int** to a **char** before searching for it within *string*.

strchr returns a pointer to the first occurrence of *character* within *string*. If *character* is not found, it returns NULL.

Having **strchr** search for a null character will always produce a pointer to the end of a string. For example,

```
char *string;
assert(strchr(string, '\0') == string + strlen(string));
```

will never fail.

Example

The following example creates functions called **replace** and **trim**. **replace** finds and replaces every occurrence of an item within a string and returns the altered string. **trim** removes all trailing spaces from a string, and returns a pointer to the altered string.

```
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <stdio.h>

char *
replace(char *string, char item, char newitem)
{
      char *start;

      /* replacing 0 is too dangerous */
      if ((start = string) == NULL || item == '\0')
            return(start);
      while ((string = strchr(string, item)) != NULL)
            *string = newitem;
      return(start);
}

char *
trim(char * str)
{
      register char *endp;

      if(str == NULL)
            return(str);

      /* start at end of string while in string and spaces */
      for(endp  = strchr(str, '\0');
          endp != str && *--endp == ' '; )
            *endp = '\0';
      return(str);
}

char string1[] = "Remove trailing spaces         ";
char string2[] = "Spaces become dashes.";
main(void)
{
      printf("\"%s\"\n", trim(string1));
      printf("%s\n", replace(string2, ' ', '-'));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.2
The C Programming Language, ed. 2, p. 249

LEXICON

See Also

index, memchr, strcspn, string handling, strpbrk, strrchr, strspn, strstr, strtok

strcmp() — String handling (libc)

Compare two strings
#include <string.h>
int strcmp(const char **string1***, const char** **string2***);**

strcmp lexicographically compares the string pointed to by *string1* with the one pointed to by *string2*. Comparison ends when a null character is encountered.

strcmp compares the two strings character by character until it finds a pair of characters that are not identical. It returns a number less than zero if the character in *string1* is less (i.e,. occurs earlier in the character table) than its counterpart in *string2*. It returns a number greater than zero if the character in *string1* is greater (i.e,. occurs later in the character table) than its counterpart in *string2*. If no characters are found to differ, then the strings are identical and **strcmp** returns zero.

Example

For an example of this function, see **fflush**.

Cross-references

Standard, §4.11.4.2
The C Programming Language, ed. 2, p. 250

See Also

memcmp, strcmp, strcoll, string handling, strncmp, strxfrm

Notes

strcmp differs from the memory-comparison routine **memcmp** in the following ways:

First, **strcmp** compares strings rather than areas of memory; therefore, it stops when it encounters a null character.

Second, **memcmp** takes two pointers to **void**, whereas **strcmp** takes two pointers to **char**. The following code illustrates how this difference affects these functions:

```
char carray[10];
int iarray[10];
char *s = "hi";
   . . .
strcmp(carray, s)          /* RIGHT */
memcmp(carray, s, 3)       /* RIGHT */
strcmp(iarray, s)          /* WRONG, 1st arg not char * */
memcmp(iarray, s, 3)       /* RIGHT, args cast to void * */
```

It is wrong to use **strcmp** to compare an **int** array with a **char** array, because this function compares strings. Using **memcmp** to compare an **int** array with a **char** array is permissible because **memcmp** simply compares areas of data.

strcoll() — String handling (libc)

Compare two strings, using locale-specific information
#include <string.h>
int strcoll(const char **string1***, const char** **string2***);**

strcoll lexicographically compares the string pointed to by *string1* with one pointed to by *string2*. Comparison ends when a null character is read. **strcoll** differs from *strcmp* in that it uses information concerning the program's locale, as set by the function **setlocale**, to help compare

strings. It can be used to provide locale-specific collating. See **localization** for more information about setting a program's locale.

strcoll compares the two strings character by character until it finds a pair of characters that are not identical. It returns a number less than zero if the character in *string1* is less (i.e,. occurs earlier in the character table) than its counterpart in *string2*. It returns a number greater than zero if the character in *string1* is greater (i.e,. occurs later in the character table) than its counterpart in *string2*. If no characters are found to differ, then the strings are identical and **strcoll** returns zero.

Cross-references

Standard, §4.11.4.3
The C Programming Language, ed. 2, p. 250

See Also

localization, memcmp, strcmp, string handling, strncmp, strxfrm

Notes

The string-comparison routines **strcoll**, **strcmp**, and **strncmp** differ from the memory-comparison routine **memcmp** in that they compare strings rather than regions of memory. They stop when they encounter a null character, but **memcmp** does not.

strcpy() — String handling (libc)

Copy one string into another
#include <string.h>
char ***strcpy(char** **string1**, **const char** ***string2**);

strcpy copies the string pointed to by *string2*, including the null character, into the area pointed to by *string1*.

strcpy returns *string1*.

Example

For an example of this function, see **realloc**.

Cross-references

Standard, §4.11.2.3
The C Programming Language, ed. 2, p. 249

See Also

memcpy, memset, string handling, strncpy

Notes

If the region of memory pointed to by *string1* overlaps with the string pointed to by *string2*, the behavior of **strcpy** is undefined.

string1 should point to enough reserved memory to hold *string2*, or code or data will be overwritten.

strcspn() — String handling (libc)

Return length a string excludes characters in another
#include <string.h>
size_t strcspn(const char ***string1**, **const char** ***string2**);

strcspn compares *string1* with *string2*. It then returns the length, in characters, for which *string1* consists of characters *not* found in *string2*.

Example

The following example returns a pointer to the first white-space character in a string. White space is defined as space, tab, or newline.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

char *
nextwhite(char *string)
{
      size_t skipcount;

      if(string == NULL)
            return NULL;
      skipcount = strcspn(string, "\t \n");
      return(string + skipcount);
}

char string1[] = "My love is like a red, red, rose";

main(void)
{
      printf(nextwhite(string1));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.3
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, string handling, strpbrk, strrchr, strspn, strstr, strtok

stream — Definition

The term *stream* is a metaphor for the flow of data between a C program and either an external I/O device (e.g., a terminal) or a file stored on a semi-permanent medium (e.g., disk or tape). A program can read data from a stream, write data into it, or (in the case of a file) directly access any named portion of it.

The Standard describes two types of stream: the *binary* stream and the *text* stream.

A binary stream is simply a sequence of bytes. The Standard requires that once a program has written a sequence of bytes into a stream, it should be able to read back the same sequence of bytes unchanged from that stream — with the sole exception that, in some environments, one or more null characters may be appended to the end of the sequence.

A text stream, on the other hand, consists of characters that have been organized into lines. A *line* in turn, consists of zero or more characters terminated by a newline character. Under MS-DOS, a text stream is practically identical to a binary stream, with the exception that it cannot read or write characters other than alphanumeric characters, the null character, and the newline character.

The Standard mandates that when data are written into a binary file, the file is not truncated. Under **Let's C**, the same is true for text files.

The Standard also mandates that an implementation should be able to handle a line that is **BUFSIZ** characters long, which includes the terminating newline character. **BUFSIZ** is a macro that is defined in the header **stdio.h**, and must be defined to be equal to at least 256.

The maximum number of streams that can be opened at any one time is given by the macro

FOPEN_MAX. Under **Let's C**, this is 20, including **stdin**, **stdout**, and **stderr**.

Cross-references

Standard, §4.9.2
The C Programming Language, ed. 2, p. 241

See Also

buffer, file, line, STDIO, stdio.h

strerror() — String handling (libc)

Translate an error number into a string
#include <string.h>
char *strerror(int *error***);**

strerror helps to generate an error message. It takes the argument *error*, which presumably is an error code generated by an error condition in a program, and may return a pointer to the corresponding error message.

The error numbers recognized and the texts of the corresponding error messages all depend upon the implementation.

Example

This example prints the user's error message and the standard error message before exiting.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stddef.h>

fatal(char * msg)
{
      int save;

      save = errno;
      /* this may clobber errno */
      fprintf(stderr, "%s", msg);
      if (save)
            fprintf(stderr, ": %s", strerror(save));
      fprintf(stderr, "\n");
      exit(save);
}

main(void)
{
      /* guaranteed wrong */
      sqrt(-1.0);
      fatal("What does sqrt say to -1?");
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.6.2
The C Programming Language, ed. 2, p. 250

See Also

error codes, errors, perror, string handling

LEXICON

Notes

strerror returns a pointer to a static array that may be overwritten by a subsequent call to **strerror**.

strerror differs from the related function **perror** in the following ways: **strerror** receives the error number through its argument *error*, whereas **perror** reads the global constant **errno**. Also, **strerror** returns a pointer to the error message, whereas **perror** writes the message directly into the standard error stream.

The error numbers recognized and the texts of the messages associated with each error number depend upon the implementation. However, **strerror** and **perror** return the same error message when handed the same error number.

strftime() — Time function (libc)

Format locale-specific time
#include <time.h>
size_t strftime(char **string***, size_t** *maximum***, const char** **format***,**
const struct tm **brokentime***);**

The function **strftime** provides a locale-specific way to print the current time and date. It also gives you an easy way to shuffle the elements of date and time into a string that suits your preferences.

strftime references the portion of the locale that is affected by the calls

```
setlocale(LC_TIME, locale);
```

or

```
setlocale(LC_ALL, locale);
```

For more information on setting locales, see the entry for **localization**.

string points to the region of memory into which **strftime** writes the date and time string it generates. *maximum* is the maximum number of characters that can be written into *string*. *string* should point to an area of allocated memory at least *maximum*+1 bytes long; if it does not, reserved portions of memory may be overwritten.

brokentime points to a structure of type **tm**, which contains the broken-down time. This structure must first be initialized by either of the functions **localtime** or **gmtime**.

Finally, *format* points to a string that contains one or more conversion specifications, which guide **strftime** in building its output string. Each conversion specification is introduced by the percent sign '%'. When the output string is built, each conversion specification is replaced by the appropriate time element. Characters within *format* that are not part of a conversion specification are copied into *string*; to write a literal percent sign, use "%%".

strftime recognizes the following conversion specifiers:

a The locale's abbreviated name for the day of the week.

A The locale's full name for the day of the week.

b The locale's abbreviated name for the month.

B The locale's full name for the month.

c The locale's default representation for the date and time.

d The day of the month as an integer (01 through 31).

H The hour as an integer (00 through 23).

I The hour as an integer (01 through 12).

j The day of the year as an integer (001 through 366).

m The month as an integer (01 through 12).

M The minute as an integer (00 through 59).

p The locale's way of indicating morning or afternoon (e.g, in the United States, "AM" or "PM").

S The second as an integer (00 through 59).

U The week of the year as an integer (00 through 53); regard Sunday as the first day of the week.

w The day of the week as an integer (0 through 6); regard Sunday as the first day of the week.

W The day of the week as an integer (0 through 6); regard Monday as the first day of the week.

x The locale's default representation of the date.

X The locale's default representation of the time.

y The year within the century (00 through 99).

Y The full year, including century.

Z The name of the locale's time zone. If no time zone can be determined, print a null string.

Use of any conversion specifier other than the ones listed above will result in undefined behavior.

If the number of characters written into *string* is less than or equal to *maximum*, then **strftime** returns the number of characters written. If, however, the number of characters to be written exceeds *maximum*, then **strftime** returns zero and the contents of the area pointed to by *string* are indeterminate.

Cross-references

Standard, §4.12.3.5
The C Programming Language, ed. 2, p. 256

See Also

asctime, ctime, date and time, gmtime, localtime, time_t, tm

Notes

strftime is modelled after the UNIX command **date**.

string.h — Header

#include <string.h>
string.h is the header that holds the declarations and definitions of all routines that handle strings and buffers. For a list of these routines, see **string handling**.

Cross-references

Standard, §4.11
The C Programming Language, ed. 2, p. 249

See Also

header, string handling

LEXICON

string handling — Overview

#include <string.h>

The Standard describes 22 routines for handling strings and regions of memory. All are declared in the header **string.h**.

String comparison

memcmp	Compare two regions
strcmp	Compare two strings
strcoll	Compare two strings, using locale information
strncmp	Compare one string with first *n* bytes of another
strxfrm	Transform a string using locale information

String concatenation

strcat	Concatenate two strings
strncat	Concatenate one string with *n* bytes of another

String copying

memcpy	Copy one region into another
memmove	Copy one region into another with which it may overlap
strcpy	Copy one string into another
strncpy	Copy *n* bytes from one string into another

String miscellaneous

memset	Fill a region with a character
strerror	Return the text of a pre-defined error message
strlen	Return the length of a string

String searching

memchr	Find first occurrence of a character in a region
strchr	Find first occurrence of a character in a string
strcspn	Find how much of the initial portion of a string consists of characters *not* found in another string
strpbrk	Find first occurrence in one string of any character from another string
strrchr	Find *last* occurrence of a character within a string
strspn	Find how much of the initial portion of string consists only of characters from another string
strstr	Find one string within another string
strtok	Break a string into tokens

Cross-references

Standard, §4.11
The C Programming Language, ed. 2, p. 249

See Also

Library, string, string.h

Notes

Let's C includes three additional functions for string searching: **index**, **pnmatch**, and **rindex**.

index and **rindex** are synonymous with, respectively, **strchr** and **strrchr**. They are included only to support existing code, and it is recommended that they not be used in new code. **pnmatch** resembles **strstr**, except that it allows you to include wildcards in the search pattern. See their respective Lexicon entries for more information.

string literal — Definition

A *string literal* consists of zero or more characters that are enclosed by quotation marks “”. For example, the following is a string literal:

```
"This is a string literal."
```

Each character within a string literal is handled exactly as if it were within a character constant, with the following exceptions: The apostrophe ´ may be represented either by itself or by the escape sequence \´, and the quotation mark “” must be represented by the escape sequence \".

A string literal has **static** duration. Its type is array of **char** which is initialized to the string of characters enclosed within the quotation marks.

If string literals are adjacent, the translator will concatenate them. For example, the string literals

```
"Here's a string literal" "Here's another string literal"
```

are automatically concatenated into one string literal.

If a string literal is not followed by another string literal, then the translator appends a null character to the end of the string as a terminator.

If two or more string literals within the same scope are identical, then the translator may store only one of them in memory and redirect to that one copy all references to any of the duplicate literals. For this reason, a program's behavior is undefined whenever it modifies a string literal.

A *wide-character literal* is a string literal that is formed of wide characters rather than ordinary, one-byte characters. It is marked by the prefix ‘L’. For example, the following

```
L"This is a wide-character literal"
```

is stored in the form of a string of wide characters. See **multibyte characters** for more information about wide characters.

Cross-references

Standard, §3.1.4
The C Programming Language, ed. 2, p. 194

See Also

", escape sequences, lexical elements, string, trigraphs

Notes

Because trigraph sequences are interpreted in translation phase 1, before string literals are parsed, a string literal that contains trigraph sequences will be translated to a different string. This is a quiet change that may break existing code.

strip — Command

Strip debug table from executable file
strip -drs *file ...*

strip removes the debug tables from a executable file that had been compiled with the **-VCSD** option. It makes the executable file noticeably smaller.

See Also

cc, commands, nm, size

Notes

strip can be used only on fully linked files.

LEXICON

strlen() — String handling (libc)

Measure the length of a string
size_t strlen(const char **string***)**

strlen counts the number of characters in *string* up to the null character that ends it. It returns the number of characters in *string*, excluding the null character that ends it.

Example

The following example prints the length of an entered string.

```
#include <stddef.h>
#include <string.h>
#include <stdio.h>

main(void)
{
      char buf[132];

      printf("Enter something\n");
      if(gets(buf) != NULL)
            printf("You entered %lu characters\n",
                  (unsigned long)strlen(buf));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.6.3
The C Programming Language, ed. 2, p. 250

See Also

string handling

strncat() — String handling (libc)

Append *n* characters of one string onto another
#include <string.h>
char *strncat(char **string1***, const char** **string2***, size_t** *n***);**

strncat copies up to *n* characters from the string pointed to by *string2* onto the end of the one pointed to by *string1*. It stops when *n* characters have been copied or it encounters a null character in *string2*, whichever occurs first. The null character at the end of *string1* is overwritten by the first character of *string2*.

strncat returns the pointer *string1*.

Example

The following example concatenates two strings to make a file name. It works for an operating system in which a file name can have no more than eight characters, and a suffix of no more than three characters.

```
#include <string.h>
#include <stdio.h>
```

```
char *
dosfilen(char *dosname, char *filename, char *filetype)
{
      *dosname = '\0';
      /* strncpy() doesn't guarantee a NULL */
      strncat(dosname, filename, 8);
      strcat(dosname, ".");
      return(strncat(dosname, filetype, 3));
}

main(void)
{
      char dosname[13];

      puts(dosfilen(dosname, "A_LONG_FILENAME",
            "A_LONG_FILETYPE"));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.3.2
The C Programming Language, ed. 2, p. 250

See Also

strcat, string handling

Notes

strncat always appends a null character onto the end of the concatenated string. Therefore, the number of characters appended to the end of *string1* could be as many as *n*+1. *string1* should point to enough allocated memory to hold itself plus *n*+1 characters; if it does not, data or code will be overwritten.

strncmp() — String handling (libc)

Compare one string with a portion of another
#include <string.h>
int strncmp(const char ******string1***, const char** ******string2***, size_t** ***n*****);**

strncmp compares *string1* with *n* bytes of *string2*. Comparison ends when a null character is read.

strncmp compares the two strings character by character until it finds a pair of characters that are not identical. It returns a number less than zero if the character in *string1* is less (i.e,. occurs earlier in the character table) than its counterpart in *string2*. It returns a number greater than zero if the character in *string1* is greater (i.e,. occurs later in the character table) than its counterpart in *string2*. If no characters are found to differ, then the strings are identical and **strncmp** returns zero. Comparison ends either when *n* bytes have been compared or a null character has been encountered in either string. The null character is compared before **strncmp** terminates.

Example

The following example searches for a word within a string. It is a simple implementation of the function **strstr**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
void fatal(const char *string)
{
      fprintf(stderr, "%s\n", string);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      int word, string, i;

      if (--argc != 2)
            fatal("Usage: example word string");

      word = strlen(argv[1]);
      string = strlen(argv[2]);
      if (word >= string)
            fatal("Word is longer than string being searched.");

      /* walk down "string" and search for "word" */
      for (i = 0; i < string - word; i++)
            if (strncmp(argv[2]+i, argv[1], word) == 0) {
                  printf("%s is in %s.\n", argv[1], argv[2]);
                  exit(EXIT_SUCCESS);
            }

      /* if we get this far, "word" isn't in "string" */
      printf("%s is not in %s.\n", argv[1], argv[2]);
      exit(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.4.4
The C Programming Language, ed. 2, p. 250

See Also

memcmp, strcmp, strcoll, string handling, strxfrm

Notes

The string-comparison routines **strcoll**, **strcmp**, and **strncmp** differ from the memory-comparison routine **memcmp** in that they compare strings rather than regions of memory. They stop when they encounter a null character, but **memcmp** does not.

strncpy() — String handling (libc)

Copy one string into another
#include <string.h>
char *strncpy(char ***_string1_, const char ***_string2_**, size_t** _n_**);**

strncpy copies *n* characters from the string pointed to by *string2* into the area pointed to by *string1*. Copying ends when *n* bytes have been copied or a null character is encountered in *string2*.

If *string2* is less than *n* characters long, **strncpy** pads *string1* with null characters until *n* characters have been deposited.

strncpy returns *string1*.

Example

This example reads a file of names and changes them from the format

```
first_name [middle_initial] last_name
```

to the format:

```
      last_name, first_name [middle_initial]
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NNAMES 512
#define MAXLEN 60
#define PERIOD '.'
#define SPACE ' '
#define COMMA ','
#define NEWLINE '\n'

char *array[NNAMES];
char gname[MAXLEN], lname[MAXLEN];

main(int argc, char *argv[])
{
      FILE *fp;
      int count, num;
      char  *name, string[MAXLEN], *cptr, *eptr;
      unsigned glength, length;

      /* check number of arguments */
      if (--argc != 1) {
            fprintf (stderr, "Usage: example filename\n");
            exit(EXIT_FAILURE);
      }

      /* open file */
      if ((fp = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }
      count = 0;

      /* get line and examine it */
      while (fgets(string, MAXLEN, fp) != NULL) {
            if ((cptr = strchr(string, PERIOD)) != NULL) {
                  cptr++;
                  cptr++;
            } else if ((cptr=strchr(string, SPACE))!=NULL)
                  cptr++;
            else continue;

            strcpy(lname, cptr);
            eptr = strchr(lname, NEWLINE);
            *eptr = COMMA;

            strcat(lname, " ");
            glength = (unsigned)(strlen(string)-strlen(cptr));
            strncpy(gname, string, glength);

            name = strncat(lname, gname, glength);
            length = (unsigned)strlen(name);
            array[count] = (char *)malloc(length + 1);

            strcpy(array[count],name);
            count++;
      }
```

LEXICON

```
      for (num = 0; num < count; num++)
            printf("%s\n", array[num]);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.2.4
The C Programming Language, ed. 2, p. 249

See Also

memcpy, memset, strcpy, string handling

Notes

string1 should point to enough reserved memory to hold *n* characters. Otherwise, code or data will be overwritten.

If the region of memory pointed to by *string1* overlaps with the string pointed to by *string2*, then the behavior of **strncpy** is undefined.

strpbrk() — String handling (libc)

Find first occurrence of a character from another string
#include <string.h>
char *strpbrk(const char ****string1*, const char ****string2***);**

strpbrk returns a pointer to the first character in *string1* that matches any character in *string2*. It returns NULL if no character in *string1* matches a character in *string2*. The set of characters that *string2* points to is sometimes called the "break string". For example,

```
      char *string = "To be, or not to be: that is the question.";
      char *brkset = ",;";
      strpbrk(string, brkset);
```

returns the value of the pointer **string** plus six. This points to the comma, which is the first character in the area pointed to by **string** that matches any character in the string pointed to by **brkset**.

Example

This example finds the first white-space character or punctuation character in a string and returns a pointer to it. White space is defined as tab, space, and newline. Punctuation is defined as the following characters:

```
      ! @ # $ % ^ & * ( ) - + = ‘ ~
      { } [ ] : ; ’ " | / , . ?
```

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

char *
findseparator(char *string)
{
      static char separators[] =
            " \n\t!@#$%^&*()-+=‘’~{}[]:;\"|\\/,.?";

      if(string == NULL)
            return(NULL);
```

```
      return strpbrk(string, separators);
}

char string1[]="I shall arise and go now/And go to Innisfree."

main(void)
{
      printf(findseparator(string1));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.4
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, strcspn, string handling, strpbrk, strrchr, strspn, strstr, strtok

Notes

strpbrk resembles the function **strtok** in functionality, but unlike **strtok**, it preserves the contents of the strings being compared. It also resembles the function **strchr**, but lets you search for any one of a group of characters, rather than for one character alone.

strrchr() — String handling (libc)

Search for rightmost occurrence of a character in a string
#include <string.h>
char *strrchr(const char ****string*, int** *character***);**

strrchr looks for the last, or rightmost, occurrence of *character* within *string*. *character* is declared to be an **int**, but is handled within the function as a **char**. Another way to describe this function is to say that it performs a reverse search for a character in a string. It is equivalent to the non-ANSI function **rindex**.

strrchr returns a pointer to the rightmost occurrence of *character*, or NULL if *character* could not be found within *string*.

Example

This example truncates a string by replacing the character after the last terminating character with a zero. It returns the truncated string.

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

char *
truncate(char *string, char endat)
{
      char *endchr;

      if(string!=NULL && (endchr=strrchr(string, endat))!=NULL)
            *++endchr = '\0';
      return(string);
}

char string1[] = "Here we go gathering nuts in May.";
```

```
main(void)
{
      puts(truncate(string1, ','));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.5
The C Programming Language, ed. 2, p. 249

See Also

memchr, rindex, strchr, strcspn, string handling, strpbrk, strspn, strstr, strtok

strspn() — String handling (libc)

Return length a string includes characters in another
#include <string.h>
size_t strspn(const char **string1***, const char** **string2***);**

strspn returns the length for which *string1* initially consists only of characters that are found in *string2*. For example,

```
char *s1 = "hello, world";
char *s2 = "kernighan & ritchie";
strcspn(s1, s2);
```

returns two, which is the length for which the first string initially consists of characters found in the second.

Example

This example returns a pointer to the first non-white-space character in a string. White space is defined as a space, tab, or newline character.

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

char *
skipwhite(char *string)
{
      size_t skipcount;

      if (string == NULL)
            return NULL;
      skipcount = strspn(string, "\t \n");
      return(string+skipcount);
}

char string1[] = "\t  Inventor: One who makes an intricate\n";
char string2[] = "arrangement of wheels, levers, and springs,\n;
char string3[] = "       and calls it civilization.\n";

main(void)
{
      printf("%s", skipwhite(string1));
      printf("%s", skipwhite(string2));
      printf("%s", skipwhite(string3));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.6
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, strcspn, string handling, strpbrk, strrchr, strstr, strtok

strstr() — String handling (libc)

Find one string within another
#include <string.h>
char *strstr(const char ****string1*, const char ****string2***);**

strstr looks for *string2* within *string1*. The terminating null character is not considered part of *string2*.

strstr returns a pointer to where *string2* begins within *string1*, or NULL if *string2* does not occur within *string1*.

For example,

```
char *string1 = "Hello, world";
char *string2 = "world";
strstr(string1, string2);
```

returns **string1** plus seven, which points to the beginning of **world** within **Hello, world**. On the other hand,

```
char *string1 = "Hello, world";
char *string2 = "worlds";
strstr(string1, string2);
```

returns NULL because **worlds** does not occur within **Hello, world**.

Example

This function counts the number of times a pattern appears in a string. The occurrences of the pattern can overlap.

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

size_t
countpat(char *string, char *pattern)
{
      size_t found_count = 0;
      char *found;

      if((found = string)==NULL || pattern==NULL)
            return 0;

      while((found = strstr(found, pattern)) != NULL) {
            /* move past beginning of this one */
            found++;
            /* count it */
            found_count++;
      }
      return(found_count);
}
```

LEXICON

```
char string1[] = "Badges, Badges -- we need no stinking Badges.";
char string2[] = "Badges";

main(void)
{
      printf("%s occurs %d times in %s\n",
            string2, countpat(string1, string2), string1);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.7
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, strcspn, string handling, strpbrk, strrchr, strspn, strtok

strtod() — General utility (libc)

Convert string to floating-point number
#include <stdlib.h>
double strtod(const char ***string*, char** ****tailptr*);**

strtod converts the string pointed to by *string* to a double-precision floating-point number.

strtod reads the string pointed to by *string*, and parses it into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that will be converted into a floating-point number. It begins when **strtod** reads a sign character, a numeral, or a decimal-point character. It can include at least one numeral, at most one decimal point, and may end with an exponent marker (either 'e' or 'E') followed by an optional sign and at least one numeral. Reading continues until **strtod** reads either a second decimal-point character or exponent marker, or any other non-numeral.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtod ignores the beginning portion of the string. It then converts the subject sequence to a double-precision number and returns it. Finally, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtod returns the **double** generated from the subject sequence. If no subject sequence could be recognized, it returns zero. If the number represented by the subject sequence is too large to fit into a **double**, then **strtod** returns **HUGE_VAL** and sets the global constant **errno** to **ERANGE**. If the number represented by the subject sequence is too small to fit into a **double**, then **strtod** returns zero and again sets **errno** to **ERANGE**.

Example

For an example of using this function in a program, see **sqrt**.

Cross-references

Standard, §4.10.4
The C Programming Language, ed. 2, p. 251

See Also

atof, atoi, atol, errno, general utilities, strtol, strtoul

Notes

The character that **strtod** recognizes as representing the decimal point depends upon the program's locale, as set by the function **setlocale**. See **localization** for more information.

Initial white space in the string pointed to by *string* is ignored. White space is defined as being all characters so recognized by the function **isspace**; the current locale setting may affect the operation of **isspace**.

strtok() — String handling (libc)

Break a string into tokens
#include <string.h>
char ***strtok(char** ****string1*****, const char** ****string2*****);**

strtok helps to divide a string into a set of tokens. *string1* points to the string to be divided, and *string2* points to the character or characters that delimit the tokens.

strtok divides a string into tokens by being called repeatedly.

On the first call to **strtok**, *string1* should point to the string being divided. **strtok** searches for a character that is *not* included within *string2*. If it finds one, then **strtok** regards it as the beginning of the first token within the string. If one cannot be found, then **strtok** returns NULL to signal that the string could not be divided into tokens. When the beginning of the first token is found, **strtok** then looks for a character that *is* included within *string2*. When one is found, **strtok** replaces it with a null character to mark the end of the first token, stores a pointer to the remainder of *string1* within a static buffer, and returns the address of the beginning of the first token.

On subsequent calls to **strtok**, set *string1* to NULL. **strtok** then looks for subsequent tokens, using the address that it saved from the first call. With each call to **strtok**, *string2* may point to a different delimiter or set of delimiters.

Example

The following example breaks **command_string** into individual tokens and puts pointers to the tokens into the array **tokenlist[]**. It then returns the number of tokens created. No more than **maxtoken** tokens will be created. **command_string** is modified to place '\0' over token separators. The token list points into **command_string**. Tokens are separated by spaces, tabs, commas, semicolons, and newlines.

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

tokenize(char *command_string, char *tokenlist[],
      size_t maxtoken)
{
      static char tokensep[]="\t\n ,;";
      int tokencount;
      char *thistoken;

      if(command_string == NULL || !maxtoken)
            return 0;

      thistoken = strtok(command_string, tokensep);

      for(tokencount = 0; tokencount < maxtoken &&
                  thistoken != NULL;) {
            tokenlist[tokencount++] = thistoken;
            thistoken = strtok(NULL, tokensep);
      }
```

LEXICON

```
      tokenlist[tokencount] = NULL;
      return tokencount;
}

#define MAXTOKEN 100
char *tokens[MAXTOKEN];
char buf[80];

main(void)
{
      for(;;) {
            int i, j;

            printf("Enter string ");
            fflush(stdout);
            if(gets(buf) == NULL)
                  exit(EXIT_SUCCESS);

            i = tokenize(buf, tokens, MAXTOKEN);
            for(j = 0; j < i; j++)
                  printf("%s\n", tokens[j]);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.8
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, strcspn, string handling, strpbrk, strrchr, strspn, strstr

strtol() — General utility (libc)

Convert string to long integer
#include <stdlib.h>
long strtol(const char **sptr*, **char** ***tailptr*, **int** *base*);

strtol converts the string pointed to by *sptr* into a **long**.

base gives the base of the number being read, from 0 to 36. This governs the form of the number that **strtol** expects. If *base* is zero, then **strtol** expects a number in the form of an integer constant. See **integer constant** for more information. If *base* is set to 16, then the string to be converted may be preceded by **0x** or **0X**.

strtol reads the string pointed to by *sptr* and parses it into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that will be converted into a **long**. It is introduced by a sign character, a numeral, or an alphabetic character appropriate to the base of the number being read. For example, if *base* is set to 16, then **strtol** will recognize the alphabetic characters 'A' through 'F' and 'a' to 'f' as indicating numbers. It continues to scan until it encounters any alphabetic character outside the set recognized for the setting of *base*, or the null character.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtol ignores the beginning portion of the string. It then converts the subject sequence to a **long**. Finally, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtol returns the **long** that it has built from the subject sequence. If it could not build a number, for whatever reason, it returns zero. If the number it builds is too large or too small to fit into a **long**, it returns, respectively, **LONG_MAX** or **LONG_MIN** and sets the global variable **errno** to the value of the macro **ERANGE**.

Cross-references

Standard, §4.10.1.5
The C Programming Language, ed. 2, p. 252

See Also

atof, atoi, atol, errno, general utility, strtod, strtoul

Notes

Initial white space in the string pointed to by *string* is ignored. White space is defined as being all characters so recognized by the function **isspace**; the current locale setting may affect the operation of **isspace**.

strtoul() — General utility (libc)

Convert string to unsigned long integer
#include <stdlib.h>
unsigned long strtoul(const char **sptr***, **char** ***tailptr***, **int** *base***);**

strtoul converts the string pointed to by *sptr* into an **unsigned long**.

base gives the base of the number being read, from 0 to 36. This governs the form of the number that **strtoul** expects. If *base* is zero, then **strtoul** expects a number in the form of an integer constant. See **integer constant** for more information. If *base* is set to 16, then the string to be converted may be preceded by **0x** or **0X**.

strtoul reads the string pointed to by *sptr* and parses it into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that will be converted into an **unsigned long**. It is introduced by a sign character, a numeral, or an alphabetic character appropriate to the base of the number being read. For example, if *base* is set to 16, then **strtoul** will recognize the alphabetic characters 'A' through 'F' and 'a' to 'f' as indicating numbers. It continues to scan until it encounters any alphabetic character outside the set recognized or the setting of *base*, or the null character.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtoul ignores the beginning portion of the string. It then converts the subject sequence to an **unsigned long**. Finally, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtoul returns the **unsigned long** that it has built from the subject sequence. If it could not build a number, for whatever reason, it returns zero. If the number it builds is too large to fit into an **unsigned long**, it returns **ULONG_MAX** and sets the global variable **errno** to the value of the macro **ERANGE**.

Example

This example uses **strtoul** as a hash function for table lookup. It demonstrates both hashing and linked lists. Hash-table lookup is the most efficient when used to look up entries in large tables; this is an example only.

LEXICON

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 * For fastest results, use a prime about 15% bigger
 * than the table. If short of space, use a smaller prime.
 */
#define HASHP 11
struct symbol {
      struct symbol *next;
      char *name;
      char *descr;
} *hasht[HASHP], codes[] = {

      NULL,        "a286",           "frogs togs",
      NULL,        "xy7800",         "doughnut holes",
      NULL,        "z678abc",        "used bits",
      NULL,        "xj781",          "black-hole varnish",
      NULL,        "h778a",          "table hash",
      NULL,        "q167",           "log(-5.2)",
      NULL,        "18888",          "quid pro quo",
      NULL,        NULL,             NULL    /* end marker */
};

void
buildTable(void)
{
      long h;
      register struct symbol *sym, **symp;

      for(symp = hasht; symp != (hasht + HASHP); symp++)
            *symp = NULL;

      for(sym = codes; sym->descr != NULL; sym++) {
            /*
             * hash by converting to base 36. There are
             * many ways to hash, but use all the data.
             */

            h = strtoul(sym->name, NULL, 36) % HASHP;
            sym->next = hasht[h];
            hasht[h] = sym;
      }
}

struct symbol *
lookup(char *s)
{
      long h;
      register struct symbol *sym;

      h = strtoul(s, NULL, 36) % HASHP;
      for(sym = hasht[h]; sym != NULL; sym = sym->next)
            if(!strcmp(sym->name, s))
                  return(sym);
      return(NULL);
}
```

```
main(void)
{
      char buf[80];
      struct symbol *sym;

      buildTable();
      for(;;) {
            printf("Enter name ");
            fflush(stdout);

            if(gets(buf) == NULL)
                  exit(EXIT_SUCCESS);

            if((sym = lookup(buf)) == NULL)
                  printf("%s not found\n", buf);

            else
                  printf("%s is %s\n", buf, sym->descr);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.1.5
The C Programming Language, ed. 2, p. 252

See Also

atof, atoi, atol, general utilities, strtod, strtol

Notes

This function has no historical usage, but provides greater functionality than does **strtol**.

Initial white space in the string pointed to by *string* is ignored. White space is defined as being all characters so recognized by the function **isspace**. The current locale setting may affect the operation of **isspace**.

struct — C keyword

The keyword **struct** introduces a *structure*. This is an aggregate data type that consists of a number of fields, or *members*, each of which can have its own name and type.

The members of a structure are stored sequentially. Unlike the related type **union**, the elements of a **struct** do not overlap. Thus, the size of a **struct** is the total of the sizes of all of its members, plus any bytes used for alignment (if the implementation requires them). Aligning bytes may not be inserted at the beginning of a **struct**, but may appear in its middle, or at the end. For this reason, it is incorrect to assume that any two members of a structure abut each other in memory.

Any type may be used within a **struct**, including bit-fields. No incomplete type may be used; thus, a **struct** may not contain a copy of itself, but it may contain a pointer to itself. A **struct** is regarded as incomplete until its closing '}' is read.

The members of a **struct** are stored in the order in which they are declared. Thus, a pointer to a **struct** also points to the beginning of the **struct**'s first member.

The following is an example of a structure:

```
struct person {
      char name[30];
      char st_address[25];
      char city[20];
      char state[2];
      char zip[9];
      char id_number[9];
} MYSELF;
```

This example defines a structure type **person**, as well as an instance of this type, called **MYSELF**.

Cross-references

Standard, §3.1.2.5, §3.5.2.1
The C Programming Language, ed. 2, pp. 127*ff*

See Also

alignment, member name, tag, types, union

strxfrm() — String handling (libc)

Transform a string
#include <string.h>
size_t strxfrm(char **string1***, const char** **string2***, size_t** *n***);**

strxfrm transforms *string2* using information concerning the program's locale, as set by the function **setlocale**. See **localization** for more information about setting a program's locale.

strxfrm writes up to *n* bytes of the transformed result into the area pointed to by *string1*. It returns the length of the transformed string, not including the terminating null character. The transformation incorporates locale-specific material into *string2*.

If *n* is set to zero, **strxfrm** returns the length of the transformed string.

If two strings return a given result when compared by **strcoll** before transformation, they will return the same result when compared by **strcmp** after transformation.

Cross-references

Standard, §4.11.4.5
The C Programming Language, ed. 2, p. 250

See Also

localization, memcmp, strcmp, strcoll, string handling, strncmp

Notes

If **strxfrm** returns a value equal to or greater than *n*, the contents of the area pointed to by *string1* are indeterminate.

swab() — Extended function (libc)

Swap a pair of bytes
void swab(char **src***, char** **dest***, unsigned short** *nb***);**

The ordering of bytes within a word differs from machine to machine. This may cause problems when moving binary data between machines. **swab** interchanges each pair of bytes in the array *src* that is *n* bytes long, and writes the result into the array *dest*. The length *nb* should be an even number, or the last byte will not be touched. *src* and *dest* may be the same place.

Example

This example prompts for an integer; it then prints the integer both as you entered it, and as it appears with its bytes swapped.

```
#include <stdio.h>
#include <stdlib.h>
extern void swab(char *src, char *dest, unsigned short nb);

main(void)
{
      short word;

      printf("Enter an integer: \n");
      scanf("%d", &word);
      printf("The word is 0x%x\n", word);
      swab(&word, &word, 2);
      printf("The word with bytes swapped is 0x%x\n", word);
      return(EXIT_SUCCESS);
}
```

See Also

byte ordering, extended miscellaneous

switch — C keyword

Select an entry in a table
switch (*expression* **)** *statement*

switch evaluates *expression*, jumps to the **case** label whose expression is equal to *expression*, and continues execution from there. *expression* may evaluate to any integral type, not just an **int**. Every **case** label's *expression* is cast to the type of *conditional* before it is compared with *expression*.

If no **case** expression matches *expression*, **switch** jumps to the point marked by the **default** label. If there is no default label, then **switch** does not jump and no statement is executed; execution then continues from the '}' that marks the end of the **switch** statement.

The program continues its execution from the point to which **switch** jumps, either until a **break**, **continue**, **goto**, or **return** statement is read, or until the '}' that encloses all of the **case** statements is encountered.

All **case** labels are subordinate to the closest enclosing **switch** statement. No two **case** labels can have expressions with the same value. However, if a **case** label introduces a secondary **switch** statement, then that **switch** statement's suite of **case** labels may duplicate the values used by the **case** labels of the outer **switch** statement.

Example

For an example of this statement, see **printf**.

Cross-references

Standard, §3.6.4.2
The C Programming Language, ed. 2, pp. 58*ff*

See Also

break, case, default, if, statements

Notes

It is good programming practice always to use a **default** label with a **switch** statement. There may be only one **default** label with any **switch** statement.

LEXICON

The number of **case** labels that can be included with a **switch** statement may vary from implementation to implementation. The Standard requires that every conforming implementation allow a **switch** statement to have up to at least 257 **case** labels.

The first edition of *The C Programming Language* requires that *conditional* may evaluate to an **int**. The Standard lifts this requirement: *conditional* may now be any integral type, from **short** to **unsigned long**. Every *expression* associated with a **case** label will be altered to conform to the type of *conditional*. Therefore, if a program depends upon *conditional* or any *expression* being an **int**, it may work differently under a conforming translator. This is a quiet change that may break existing code.

system() — General utility (libc)

Suspend a program and execute another
#include <stdlib.h>
int system(const char ******program***);**

system provides a way to execute another program from within a C program. It suspends the program currently being run, and passes the name pointed to by *program* to MS-DOS. When *program* has finished executing, MS-DOS returns to the current program, which then continues its operation.

If *program* is set to NULL, **system** checks to see if a command processor exists. In this case, **system** returns zero if a command processor does not exist and nonzero if it does. If *program* is set to any value other than NULL, then what **system** returns is defined by the implementation.

Example

This example execute system commands on request.

```
#include <stdio.h>
#include <stdlib.h>

syscmds(char * prompt)
{
      for(;;) {
            char buf[80];

            printf(prompt);
            fflush(stdout);
            if(gets(buf) == NULL || !strcmp(buf, "exit"))
                  return;
            system(buf);
      }
}

main(void)
{
      printf("Enter system commands: ");
      syscmds(">");
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.4.5
The C Programming Language, ed. 2, p. 253

See Also

command processor, exit, general utilities

T

tag — Definition

A *tag* is a name that follows the keywords **struct**, **union**, or **enum**. It names the type of object so declared. For example, in the following code

```
struct STR {
        . . .
};
```

the identifier **STR** is a tag. It defines a new type of structure called **STR**. It does not, however, allocate any storage for any instance of this type.

Cross-references

Standard, §3.1.2.6
The C Programming Language, ed. 2, pp. 212*ff*

See Also

member, name space

tail — Command

Print the end of a file
tail [+*n***[bcl]] [***file***]**
tail [-*n***[bcl]] [***file***]**

tail copies the last part of *file*, or of the standard input if none is named, to the standard output.

The given *number* tells **tail** where to begin to copy the data. Numbers of the form +*number* count from the beginning of the file; those of the form -*number* count from the end of the file.

A specifier of blocks, characters, or lines (*b*, **c**, or **l**, respectively) may follow the number; the default is lines. If no *number* is specified, a default of -10 is assumed.

See Also

commands, egrep

Notes

Because **tail** buffers data measured from the end of the file, large counts may not work.

tan() — Mathematics (libm)

Calculate tangent
#include <math.h>
double tan(double *radian***);**

tan calculates and returns the tangent of its argument *radian*, which must be in radian measure.

Cross-references

Standard, §4.5.2.7
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan, atan2, cos, mathematics, sin

tanh() — Mathematics (libm)

Calculate hyperbolic tangent
#include <math.h>
double tanh(double *value***);**

tanh calculates the hyperbolic tangent of *radian*.

Cross-references

Standard, §4.5.3.3
The C Programming Language, ed. 2, p. 251

See Also

cosh, mathematics, sinh

technical information — Overview

The Lexicon includes the following articles that give technical information on the IBM PC and MS-DOS:

ansi.sys	Device driver for console
BIOS data area	List magic areas within memory
byte ordering	Describe ordering of bytes
i8087	Floating-point co-processor
keyboard	Give keyboard scan codes
LARGE model	Describe Intel multi-segment memory model
model	Describe Intel memory models
SMALL model	Describe Intel single-segment memory model

See Also

DOS-specific information

tempnam() — Extended function (libc)

Generate a unique name for a temporary file
char *tempnam(char **directory***, **char** **name***);**

tempnam constructs a unique temporary name that can be used to name a file.

directory points to the name of the directory in which you want the temporary file written. If this variable is NULL, **tempnam** reads the environmental variable **TMPDIR** and uses it for *directory*. If neither *directory* nor **TMPDIR** is given, **tempnam** uses **\tmp**.

name points to the string of letters that will prefix the temporary name. This string should not be more than three or four characters, to prevent truncation or duplication of temporary file names. If *name* is NULL, **tempnam** will set it to **t**.

tempnam uses **malloc** to allocate a buffer for the temporary file name it returns. If all goes well, it returns a pointer to the temporary name it has written. Otherwise, it returns NULL if the allocation fails or if it cannot build a temporary file name successfully.

See Also

extended miscellaneous, mktemp, TMPDIR, tmpfile, tmpnam

Notes

tempnam is not described in the ANSI Standard. Programs that use it will not conform strictly the Standard, and may not be portable to other compilers or environments.

time() — Time function (libc)

Get current calendar time
#include <time.h>
time_t time(time_t ******tp***);**

The function **time** returns the current calendar time. If *tp* is set a value other than NULL, then **time** writes the result to the object pointed to by *tp*. **Let's C** defines the current system time as the number of seconds since January 1, 1970, 0h00m00s UTC.

time returns an object of the type **time_t**, which is defined in the header **time.h**. If the current calendar time is not available, **time** returns -1 cast to type **time_t**.

Example

This example displays the time.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void)
{
      time_t t;

      /* get the time */
      if(-1 == time(&t))
            printf("The time is unavailable?");
      else
            /* display it */
            printf(ctime(&t));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.2.4
The C Programming Language, ed. 2, p. 256

See Also

clock, date and time, difftime, mktime, time_t

time — Command

Print current time/Time execution of a command
time
time *command*
time " *command arguments* **"**

The command **time** performs two different tasks, depending upon whether it is used with or without arguments.

When **time** is typed without any arguments, it prints the date and time. The date and time are presented in a string of the form:

```
Thu Apr  7 10:35:53 1988 CDT
```

The extension "CDT" stands for "Central Daylight Time". Daylight savings time will be returned only if the macro **TIMEZONE** is set properly in your **profile**. See **TIMEZONE** for more information.

If **time** is used with one or more arguments, it times the execution of a command. For example, typing **time ls** prints the contents of the current directory, then prints a string of the form:

```
00:00:02.340
```

which states how long the command took to execute.

If you wish to time a command that takes arguments, you must enclose the command and its arguments within quotation marks. For example, to time how long it takes to compile the program **window.c** with the **-VGEM** option to the compiler, use the command:

```
time "cc -VGEM window.c"
```

See Also

commands, date, msh, time (overview)

time.h — Header

Header for date and time
#include <time.h>

time.h is the header that declares the function and defines the types used to represent time. It contains prototypes for the following nine functions:

asctime	Convert broken-down time into text
clock	Get processor time used by the program
ctime	Convert calendar time to text
difftime	Calculate difference between two times
gmtime	Convert calendar time to Universal Coordinated Time
localtime	Convert calendar time to local time
mktime	Convert broken-down time into calendar time
strftime	Format locale-specific time
time	Get current calendar time

It also contains definitions for the following data types:

clock_t	Encode system time
time_t	Encode calendar time
tm	Encode broken-down time

It contains a definition for the macro **CLK_TCK**, which is used to convert the value returned by the function **clock** into seconds of real time.

Cross-references

Standard, §4.12
The C Programming Language, ed. 2, p. 255

See Also

CLK_TCK, date and time, header, xtime.h

time_t — Type

Calendar time
#include <time.h>

time_t is a data type that is defined in the header **time.h**. It is an arithmetic type that can represent time.

time_t is used to hold the calendar time, as computed from the system time by the function **time**. The functions **localtime** and **gmtime** use **time_t** to generate broken-down time, and the function **ctime** uses it to create a string that states the current date and time. The function **mktime** reads broken-down time and returns calendar time of type **time_t**.

Example

For an example of using this type in a program, see **difftime**.

Cross-references

Standard, §4.12.1
The C Programming Language, ed. 2, p. 255

See Also

broken-down time, calendar time, clock_t, date and time

time_to_jday() — Extended function (libc)

Convert system time to Julian date
#include <time.h>
#include <xtime.h>
jday_t time_to_jday(time_t *time***);**

time_to_jday converts system time to Julian days. *time* is the current system time. It is declared to be of type **time_t**, which is defined in the header file **time.h** as being equivalent to a **long**. **Let's C** defines the current system time as being the number of seconds from January 1, 1970, 0h00m00s UTC. The function **time** returns the current system time in this format.

time_to_jday returns the structure **jday_t**, which is defined in the header **xtime.h**. **jday_t** consists of two **unsigned long**s. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

See Also

extended time, jday_to_time, jday_to_tm, tm_to_jday, xtime.h

Notes

To conform with the ANSI Standard, this function has been moved from the header **time.h** to the header **xtime.h**. This may require that some code be altered.

TIMEZONE — Environmental variable

Time zone information
TIMEZONE=*standard***:***offset***[:***daylight***:** *date***:***date***:***hour***:***minutes***]**

TIMEZONE is an environmental parameter that holds information about the user's time zone. This information is used by **Let's C**'s time routines to construct their description of the current time and day.

To set **TIMEZONE**, use the **set** command, as follows:

```
set TIMEZONE=description
```

where *description* is the string that describes your time zone. What this string consists of will be described below. Most users write this command into the file **AUTOEXEC.BAT**, so that **TIMEZONE** is set automatically whenever they reboot their system.

The Description String

A **TIMEZONE** description string consists of seven fields that are separated by colons. Fields 1 and 2 must be filled; fields 3 through 7 are optional.

Field 1 gives the name of your standard time zone. Field 2 gives the time zone's offset from Universal Coordinated Time (UTC) in minutes. Offsets are positive for time zones west of Greenwich and negative for time zones east of Greenwich. For example, users in Chicago set these fields as follows:

LEXICON

```
TIMEZONE=CST:360
```

CST is an abbreviation for Central Standard Time, that area's time zone; and 360 refers to the fact that Chicago's time zone is 360 minutes (six hours) behind that of Greenwich.

Field 3 gives the name of the local daylight saving time zone. In Chicago, for example, this field would be set as follows:

```
TIMEZONE=CST:360:CDT
```

CDT is an abbreviation for Central Daylight Time. The absence of this field indicates that your area does not use daylight saving time.

Fields 4 and 5 specify the dates on which daylight saving time begins and ends. If field 3 is set but fields 4 and 5 are not, changes between standard time and daylight saving time will be assumed to occur at the times legislated in the United States in 1986: at 2 A.M. standard time on the first Sunday in April, and at 2 A.M. daylight saving time on the last Sunday in October.

Fields 4 and 5 each consist of three numbers separated by periods. The first number specifies which occurrence of the day in the month marks the change, counting positive occurrences from the beginning of the month and negative occurrences from the the end of the month. The second number specifies a day of the week, numbering Sunday as one. The third number specifies a month of the year, numbering January as one. For example, in Chicago fields 4 and 5 are set to the following:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10
```

If the first number in either field is set to zero, then the last two numbers are assumed to indicate an absolute date. This is done because some countries switch to daylight saving time on the same day each year, instead of a given day of the week.

Finally, fields 6 and 7 specify the hour of the day at which daylight saving time begins and ends, and the number of minutes of adjustment. In Chicago, these are set as follows:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10:2:60
```

The '2' of field 6 indicates that the switch to daylight savings time occurs at 2 A.M. The "60" of field 7 indicates that daylight savings time changes the local time by 60 minutes. Although 60 minutes is the standard change, some regions of the world shift by 30, 45, 90, or 120 minutes; the last shift is also called "double daylight saving time".

For an example of this variable's use in a program, see the entry for **asctime**.

See Also

environmental variable, time

Notes

This environmental variable should be set only if you have set your computer system's time to conform with UTC. Otherwise, it will cause such functions as **localtime** to incorrectly offset the time they return.

For those requiring more information on this subject, see *Time Changes in the World*, compiled by Doris Chase Doane (three volumes, Hollywood, CA, Professional Astrologers, Inc., 1970).

tm — Type

Encode broken-down time
#include <time.h>

tm is the structure that holds the elements of broken-down time. It contains the following fields. (The values representable are shown within parentheses):

int tm_sec	Second (0-59)
int tm_min	Minute (0-59)
int tm_hour	Hour (0-23): 0 == midnight
int tm_mday	Day of the month (1-31)
int tm_mon	Month (0-11): 0 == January
int tm_year	Year since 1900 A.D.
int tm_wday	Day of week (0-6): 0 == Sunday
int tm_yday	Day of the year (0-366)
int tm_isdst	Daylight savings time flag

The field **tm_isdst** indicates whether daylight saving time is currently in effect. It is set to a positive number if daylight saving time is in effect, to zero if it is not, and to a negative number if information concerning daylight saving time is not available.

The functions **localtime** and **gmtime** read the calendar time, as returned by the function **time**, and use it initialize **tm**; they then return a pointer to the structure.

The function **strftime** reads **tm** and uses it to build strings that present the date and time in a locale-specific manner. Finally, the function **mktime** reads **tm** and uses its contents to compute the corresponding calendar time.

Example

For an example of using this structure in a program, see **localtime**.

Cross-references

Standard, §4.12.1
The C Programming Language, ed. 2, p. 255

See Also

broken-down time, calendar time, clock_t, date and time, time_t

tm_to_jday() — Extended function (libc)

Convert calendar format to Julian time
#include <time.h>
#include <xtime.h>
jday_t tm_to_jday(tm ******time***);**

tm_to_jday converts the system time, as described in the system calendar format, to Julian time.

time points to a copy of the structure **tm**, which is defined in the header **time.h**. The functions **gmtime** and **localtime** return the current time in this format. For more information on **tm**, see the entry for **time**.

tm_to_jday returns the structure **jday_t**, which is defined in the header **xtime.h**. **jday_t** to consist of two **unsigned long**s. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

LEXICON

See Also

extended time, jday_to_time, jday_to_tm, time, time_to_jday, xtime.h

Notes

To conform with the ANSI Standard, this function has been moved from the header **time.h** to the header **xtime.h**. This may require that some code be altered.

TMPDIR — Environmental variable

Directory that holds temporary files

TMPDIR names the directory into which **Let's C** writes its temporary files. If this variable is not set, the default is the directory in which the source files are kept. Note that this variable need be set only if space is a problem on the storage device that holds your current directory. For example, the command

```
set TMPDIR=a:\tmp
```

typed at the system prompt tells **cc** to write temporary files in the directory **tmp** on drive A:.

It is a good idea to set **TMPDIR** in **autoexec.bat**, to ensure that it is always set correctly.

See Also

cc, environmental variable

tmpfile() — STDIO (libc)

Create a temporary file
#include <stdio.h>
FILE *tmpfile(void);

tmpfile creates a file to hold data temporarily. The file is opened into binary update mode (**wb+**) and is removed automatically when it is closed or when the program ends. There is no way to access the temporary file by name. If your program needs to do so, it should open a file explicitly.

tmpfile returns NULL if it could not create the temporary file. If it could, it returns a pointer to the **FILE** associated with the temporary file. The function **exit** removes all files created by **tmpfile**.

Example

This example implements a primitive file editor that can edit large files. It uses two temporary files to keep all changes. The editor accepts the following commands:

dn	delete; **d52** deletes line 52
in	insert; **i7** inserts line before line 7
pn	print; **p17** prints line 17
p	print the entire file
w	write the edited file and quit
q	quit without writing the file

The entire temporary file is copied with each command.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdarg.h>

FILE *fp, *tmp[2];
int linecount;
```

```
void
fatal(char *format, ...)
{
      va_list argptr;

      /* if there is a system message, display it */
      if(errno)
            perror(NULL);

      /* if there is a user message, use it */
      if(format != NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}

/*
 * Copy up to line number or EOF.
 * Return number of lines copied.
 */
static int
copy(int line, FILE *ifp, FILE *ofp)
{
      int i, c, count;

      count = 0;
      for(c=i=1; (i<line || line==-1) && c!=EOF; i++) {
            while((c = fgetc(ifp)) != EOF && c != '\n')
                  fputc(c, ofp);

            if(c == '\n') {
                  count++;
                  fputc('\n', ofp);
            }
      }
      return(count);
}

/*
 * Read a file until line number is read.
 * Return 1 if line is found before EOF.
 */
static int
find(int line, FILE *ifp)
{
      int i, c;

      for(c=i=1; i<line && c!=EOF; i++)
            while((c = fgetc(ifp)) != EOF && c != '\n')
                  ;
      return(c != EOF);
}

main(int argc, char *argv[])
{
      int i, line, args;
      char c, cmdbuf[80];

      if(argc != 2)
            fatal("usage: tmpfile filename\n");
```

LEXICON

```
if((tmp[0]=tmpfile())==NULL||(tmp[1]=tmpfile())==NULL)
      fatal("Error opening tmpfile\n");

if((fp = fopen(argv[1], "r")) == NULL)
      fatal("Error opening %s\n", argv[1]);

linecount = copy(-1, fp, tmp[i = 0]);
fclose(fp);

/* one file pass per command */
for(;;) {
      if(gets(cmdbuf) == NULL)
            fatal("EOF on stdin\n");

      if(!(args = sscanf(cmdbuf, "%c%d", &c, &line)))
            continue;
      fseek(tmp[i], 0L, SEEK_SET);

      switch(c) {
      /* Write edited file */
      case 'w':
            if((fp = fopen(argv[1], "w")) ==  NULL)
                  fatal("Error opening %s\n", argv[1]);
            copy(linecount + 1, tmp[i], fp);
            fclose(fp);

      /* Quit */
      case 'q':
            exit(EXIT_SUCCESS);

      /* Print entire file */
      case 'p':
            if(args == 1) {
                  copy(linecount + 1, tmp[i], stdout);
                  continue;
            }
            if(find(line, tmp[i]))
                  copy(2, tmp[i], stdout);
            continue;

      /* Delete a line */
      case 'd':
            if(args == 1)
                  printf("dn where n is a number\n");
            else if(line > linecount)
                  printf("only %d lines\n", linecount);

            else {
                  copy(line, tmp[i], tmp[i^1]);
                  if(find(2, tmp[i]))
                        copy(-1, tmp[i], tmp[i^1]);

                  linecount--;
                  fseek(tmp[i], 0L, SEEK_SET);
                  i ^= 1;
            }
            continue;
```

```
            /* Insert a line */
            case 'i':
                  if(1 == args)
                        printf("in where n is a number\n");
                  else if(line > linecount)
                        printf("only %d lines\n", linecount);

                  else {
                        copy(line, tmp[i], tmp[i^1]);
                        printf("Enter inserted line\n");
                        copy(2, stdin, tmp[i^1]);
                        copy(-1, tmp[i], tmp[i^1]);
                        linecount++;

                        fseek(tmp[i], 0L, SEEK_SET);
                        i ^= 1;
                  }
                  continue;

            default:
                  printf("Invalid request\n");
                  continue;
            }
      }
}
```

Cross-references

Standard, §4.9.4.3
The C Programming Language, ed. 2, p. 243

See Also

mktemp, STDIO, tempnam, tmpnam

Notes

If a program exits abnormally or aborts, the files created by **tmpfile** may not be removed.

tmpnam() — STDIO (libc)

Generate a unique name for a temporary file
#include <stdio.h>
char *tmpnam(char *_name_);

tmpnam constructs a unique name for a file. The names returned by **tmpnam** generally are mechanical concatenations of letters, and therefore are mostly used to name temporary files, which are never seen by the user. Unlike a file created by **tmpfile**, a file named by **tmpnam** does not automatically disappear when the program exits. It must be explicitly removed before the program ends if you want it to disappear.

name points to the buffer into which **tmpnam** writes the name it generates. If *name* is set to NULL, **tmpnam** writes the name into an internal buffer that may be overwritten each time you call this function.

tmpnam returns a pointer to the temporary name. Unlike the related function **tempnam**, **tmpnam** assumes that the temporary file will be written into directory **\tmp** and builds the name accordingly.

Example

The following example uses **tmpnam** to generate some file names, opens one, and writes the rest of the names into it.

```
#include <stdio.h>
#include <stdlib.h>

void fatal(const char *string)
{
      fprintf(stderr, "%s\n", string);
      exit(EXIT_FAILURE);
}

main()
{
      int i, files;
      FILE *fp;
      char buffer[L_tmpnam];

      if ((fp = fopen(tmpnam(buffer), "w")) == NULL)
            fatal("Cannot open temporary file");
      printf("Temporary file name is %s\n", buffer);

      /* put realistic limit on number of names */
      100 > TMP_MAX ? files = TMP_MAX : files = 100;
      for(i = 0; i < files; i++)
            fprintf(fp, "%s\n", tmpnam(NULL));

      fclose(fp);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.4.4
The C Programming Language, ed. 2, p. 243

See Also

L_tmpnam, mktemp, STDIO, tempnam, tmpfile, TMP_MAX

Notes

If you want the file name to be written into *buffer*, you should allocate at least **L_tmpnam** bytes of memory for it; **L_tmpnam** is defined in the header **stdio.h**.

tmpnam can be called at least **TMP_MAX** times to return unique file names. **TMP_MAX** is also set in **stdio.h**.

toascii() — Extended macro (xctype.h)

Convert characters to ASCII
#include <xctype.h>
int toascii(int *c***);**

toascii takes any integer value *c*, keeps the low seven bits unchanged, and changes the others to zero. This, in effect, transforms the integer value to an ASCII character. **toascii** then returns the transformed integer. If *c* is a valid ASCII character, it is returned unchanged.

Example

This example prompts for a file name. It then opens the file and prints its contents, while converting all non-alphanumeric characters to alphanumeric.

```
#include <stdio.h>
#include <stdlib.h>
#include <xctype.h>
```

```
main(void)
{
      FILE *fp;
      int ch;
      int filename[20];

      printf("Enter file name: ");
      fflush(stdout);
      gets(filename);

      if ((fp = fopen(filename, "r")) != NULL) {
            while ((ch = fgetc(fp) != EOF)
                  putchar(isascii(ch) ? ch : toascii(ch));
      } else {
            printf("Cannot open %s\n", filename);
            exit(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

See Also

extended character handling

Notes

To conform to the ANSI Standard, this macro has been moved from the header **ctype.h** to the header **ctype.h**. This may require that some code be altered.

This macro is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

token — Definition

A *token* is the basic, indivisible unit of text that is processed by the translator.

There are two varieties of token: *lexical token* and *preprocessing token*. When the Standard uses the term "token," it refers to what is here called a "lexical token." Note, too, that the term "preprocessing token" does not mean a token that is manipulated only by the preprocessor.

Preprocessing tokens form the following varieties of lexical elements:

- Character constant.
- Header name.
- Identifier.
- Operator.
- Preprocessing number.
- Punctuator.
- String literal.
- Each non-white space character that does not fall into one of the above categories.

White-space characters can appear only within a header name, a character constant, or a string literal; in all other instances, white space separates tokens.

Preprocessing tokens are processed during phases 3 through 6 of translation. For details on translation, see the entry for **translation phases**. In brief, all preprocessing directives are executed: **#include** states are expanded, code is conditionally included, and macros are expanded. Each

comment is replaced with one white-space character.

Adjacent string literals are concatenated and clusters of text that are not separated by white space are parsed. A cluster of text is always parsed into the longest possible sequence of characters that forms a valid token. For example, the text

```
a+++++b
```

must be parsed into:

```
a ++ ++ + b
```

The preprocessor passes unchanged what it does not recognize as being a preprocessor token.

Lexical tokens (which the Standard calls simply "tokens") form the following types of lexical elements:

- Constant.
- Identifier.
- Keyword.
- Operator.
- Punctuator.
- String literal.

Lexical tokens are parsed, analyzed, and linked.

Cross-references

Standard, §3.1
The C Programming Language, ed. 2, pp. 191, 229

See Also

lexical elements, translation phase

tolower() — Character handling (ctype.h)

Convert character to lower case
int tolower(int *c***);**

The macro **tolower** converts the upper-case character *c* to its corresponding lower-case character, as defined by the locale's character set. The Standard defines an upper-case character as one for which the function **isupper** returns true. *c* must be a value that is representable as an **unsigned char** or **EOF**.

If *c* is an upper-case letter, then **tolower** returns the corresponding lower-case letter. If *c* is not a letter or is already lower case, then **tolower** returns it unchanged.

Example

The following example demonstrates **tolower** and **toupper**. It reverses the case of every character in a text file.

```
void fatal(const char *message)
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}
```

```
#include <ctype.h>
#include <stdio.h>
void fatal(const char *string);

main(int argc, char *argv[])
{
      FILE *fp;
      int ch;

      if (argc != 2)
            fatal("usage: example filename");

      if ((fp = fopen(argv[1], "r")) == NULL)
            fatal("cannot open text file");

      while ((ch = fgetc(fp)) != EOF)
            putchar(isupper(ch) ? tolower(ch) : toupper(ch));
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.3.2.1
The C Programming Language, ed. 2, p. 249

See Also

character handling, toupper

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

toupper() — Character handling (libc)

Convert character to upper case
int toupper(int *c***);**

toupper converts the lower-case character *c* to its corresponding upper-case character. The Standard defines an lower-case character as one for which the function **islower** returns true. *c* must be either a value that is representable as an **unsigned char** or **EOF**.

If **c** is an lower-case letter, then **toupper** returns the corresponding upper-case letter for the locale's character set. If *c* is not a letter or is already upper case, then **toupper** returns it unchanged.

Example

For an example of this function, see **tolower**.

Cross-references

Standard, §4.3.2.2
The C Programming Language, ed. 2, p. 249

See Also

_toupper, character handling, tolower

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

LEXICON

translation unit — Definition

A *translation unit* is the basic unit of code that is translated into executable form. It consists of a source file, plus all headers that are requested with the preprocessing directive **#include**, and excluding all code that is skipped by preprocessing conditional inclusion.

Cross-references

Standard, §2.1.1.1
The C Programming Language, ed. 2, p. 191

See Also

#include, conditional inclusion, Environment, source file

trigraph sequences — Definition

A *trigraph sequence* is a set of three characters that represents one character in the C character set. The set of trigraph sequences was created to allow users to use the full range of C characters, even if their keyboards do not implement the full C character set. Trigraph sequences are also useful with input devices that reserve one or more members of the C character set for internal use.

Each trigraph sequence is introduced by two question marks. The third character in the sequence indicates which character is being represented. The following table gives the set of trigraph sequences:

Trigraph Sequence	*Character Represented*
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

The characters represented are the ones used in the C character set but not included in the ISO 646 character set. ISO 646 describes an invariant sub-set of the ASCII character set.

Trigraph sequences are interpreted even if they occur within a string literal or a character constant. This is because they are interpreted before the source code is tokenized; see **translation phases** for more information. Thus, strings that uses a literal "??" will not work the same as under a non-ANSI implementation of C. For example, the function call

```
printf("Feel lucky, punk??!\n");
```

would print:

```
Feel lucky, punk|
```

This is a silent change that may break existing code.

To print a pair of questions marks, use the escape sequence '\?\?'. For example:

```
printf("Feel lucky, punk\?\?!\n");
```

Cross-references

Standard, §2.2.1.1
The C Programming Language, ed. 2, p. 229

See Also

Environment

true — Definition

In the context of a C program, an expression is *true* if it yields nonzero.

See Also

Definitions, false

typedef — C keyword

Synonym for another type

The storage-class specifier **typedef** names a synonym for a type.

The new synonym must include all qualifiers and storage-class specifiers. For example, the declaration

```
typedef volatile unsigned long int giant;
```

states that the type **giant** is a synonym for **volatile unsigned long int**. Thus, the declaration

```
giant example();
```

declares, in effect, that the function **example** returns an **volatile unsigned long int**. An object declared to be type **giant** and one declared to be type **volatile unsigned long int** behave exactly the same.

typedef is often used to declare a structure type. For example, the structure declaration

```
typedef struct {
      int member1, member2;
      long member3;
} EXAMPLE;
```

declares that **EXAMPLE** is a type name, and that it is a synonym for the structure that precedes it.

Cross-references

Standard, §3.5.6
The C Programming Language, ed. 2, p. 146

See Also

storage-class specifiers, types

Notes

The term *typedef* also describes a type that is defined in a **typedef** statement.

The Standard does not allow benign redeclarations of typedefs. For example, if the declaration

```
typedef int SINT;
```

were included in a header and the same declaration appeared in a source file that included this header, a diagnostic message should appear during translation.

type qualifier — Overview

A *type qualifier* is, as its name implies, a keyword that alters the nature of a type in a significant way.

There are two type qualifiers:

const Qualify an identifier as not modifiable
volatile Qualify an identifier as changing frequently

The changes affected by a type qualifier take effect only in expressions that yield an lvalue.

No type qualifier may modify an identifier more than once, either directly or via a **typedef**. Also, two types are considered to be compatible only if their qualifiers match.

Many quirks surround the use of qualifiers. For example:

```
const int *cip;
int *ip;

cip = ip;        /* RIGHT */
ip = cip;        /* WRONG */
```

In effect, assignments that serve to "hide" the qualified object must be diagnosed. Although the above examples uses the qualifier **const**, the same restrictions apply to any combination of qualifiers on an object.

Cross-references

Standard, §3.5.3
The C Programming Language, ed. 2, p. 211

See Also

declarations

Notes

Because type qualifiers can alter the manner in which an object is accessed, they can be considered to be "access modifiers".

types — Overview

Type determines the meaning of a value stored in an object or returned by a function. For example, if an object four bytes long were declared to be type **long**, the meaning of its contents is quite different than if it were declared to be of type **long** *, or a pointer to a **long**. In the former instance, the contents are regarded as an absolute value. In the latter, the contents are regarded as an address of another object.

The Standard organizes types into a number of varieties and categories, as follows:

Aggregate types
All array and structure types.

Arithmetic types
The set of integral and floating types.

Array types
A set of objects that have the same type and are in contiguous memory.

Basic types
The set of **char**, the signed and unsigned integer types, and the floating types; i.e., arithmetic types but not enumerated types.

Composite type
A type constructed from two compatible types, one of which has additional type information. For example, the declarations

```
int example;
   . . .
```

```
const int example;
```

together form a composite type.

Derived declarator types
The set of array, function, and pointer types.

Derived types
The set of array, function, pointer, structure, and **union** types that are derived from the basic types.

Enumerated type
A set of named integer constant values that comprise an enumeration.

Floating types
The types **float**, **double**, or **long double**.

Function types
The type that describes a given function with a specified return type and specified number and types of parameters.

Incomplete types
A type for which the translator does not possess all necessary information. Examples are an array of unknown size, or a structure or **union** of unknown content. An incomplete type must be completed by the end of translation.

Integral types
The set of type **char**, the signed and unsigned integer types, and the enumerated types.

Object types
The set of types that describe objects, rather than functions.

Pointer type
A type that describes the type of object to which a pointer points. The two classes of pointers are object pointers and function pointers. Object pointers are referred to by the type of object to which they point.

Qualified type
A type whose top type is qualified with some combination of the type qualifiers **const**, **noalias**, or **volatile**.

Scalar types
The set of arithmetic types and pointer types.

Signed integer types
Any of the types **signed char**, **int**, **long int**, or **short int**. Any of the last three types may also use the prefix **signed**, but the addition of this prefix does not change them in any way.

Structure type
A type that describes a group of data objects that are contiguous; each object may have its own specified type and its own name.

Top type
The top type of a basic type is the type itself. The top type of a derived type is the first type used to describe the type; for example, the type **int** * is described as "pointer to **int**"; therefore, its top type is pointer.

union *type*
A type that describes a set of objects that overlap in memory. Each object may have its own type and its own name.

LEXICON

Unqualified type
Any type whose top type is *not* qualified with the type qualifiers **const**, **noalias**, or **volatile**.

Unsigned integer types
Any of the types **unsigned char**, **unsigned int**, **unsigned long int**, and **unsigned short int**.

Basic Types

The following is the set of basic types. Those on the same line are synonyms:

char
double
float
int, signed int
long double
long int, long, signed long, signed long int
signed char
short int, short, signed short int, signed short
unsigned char
unsigned int
unsigned long int, unsigned long
unsigned short int, unsigned short

Data Formats

Mark Williams Company has written C compilers for a number of different computers. Each has a unique architecture and defines data formats in its own way.

The following table gives the sizes, in **char**s, of the data types as they are defined by various microprocessors.

Type	*i8086 SMALL*	*i8086 LARGE*	*Z8001*	*Z8002*	*68000*	*PDP11*	*VAX*
char	1	1	1	1	1	1	1
double	8	8	8	8	8	8	8
float	4	4	4	4	4	4	4
int	2	2	2	2	2	2	4
long	4	4	4	4	4	4	4
long double	8	8	8	8	8	8	8
pointer	2	4	4	2	4	2	4
short	2	2	2	2	2	2	2

Let's C places some alignment restrictions on data, which conform to all restrictions set by the microprocessor. Byte ordering is set by the microprocessor. See the Lexicon entry on **byte ordering** for more information.

Type Checking

C is not strongly typed, which means that it allows different types to be mixed relatively freely, and be changed (or **cast**) from one type to another.

Let's C checks types more strictly than the C standard implies. **Let's C**'s type checking can be enabled or disabled in degrees, using **-VSTRICT** and other "variant" options with the **cc** command.

Type Promotion

In arithmetic expressions, **Let's C** promotes one signed type to another signed type by sign extension, and promotes one unsigned type to another unsigned type by zero padding. For example, **char** promotes to **int** by sign extension, while **unsigned char** promotes to **unsigned int** by zero padding.

Cross-references

Standard, §3.1.2.5
The C Programming Language, ed. 2, p. 195

See Also

identifiers, signed, struct, type specifiers, union, unsigned

Notes

On some machines, **char** is a synonym for **signed char**. On others, it is a synonym for **unsigned char**. You should declare a **char** variable to be **signed** or **unsigned** if its behavior when promoted to **int** is significant.

type specifier — Overview

A *type specifier* specifies the type of an object or function when it is declared.

The following lists the legal C type specifiers:

char
double
enum *tag-name*
float
int
long
signed
short
struct *tag-name*
unsigned
union *tag-name*
void

The type specifiers can be combined into any one of the following combinations. Those on the same line are synonyms:

char
double
enum *type-name*
float
int, **signed**, **signed int**
long double
long int, **long**, **signed long**, **signed long int**
signed char
short int, **short, signed short int**, **signed short**
struct *type-name*
typedef *name*
union specifier
unsigned char
unsigned int, **unsigned**
unsigned long int, **unsigned long**
unsigned short int, **unsigned short**
void

Cross-references

Standard, §3.5.2
The C Programming Language, ed. 2, p. 211

LEXICON

See Also

types, enum, struct, typedef, union, void

U

ungetc() — STDIO (libc)

Push a character back into the input stream
#include <stdio.h>
int ungetc(int *character*, **FILE** **fp*);

ungetc converts *character* to an **unsigned char** and pushes it back into the stream pointed to by *fp*, where the next call to an input function will read it as the next character available from the stream. **ungetc** clears the end-of-file indicator for the stream.

The Standard only guarantees that one character can safely be pushed back into *fp* at any given time. A subsequent call to **fflush**, **fseek**, **fsetpos**, or **rewind** will discard the "ungotten" character.

ungetc returns *character* if it could be pushed back onto *fp*. Otherwise, it returns **EOF**. If *character* is equivalent to **EOF**, **ungetc** will fail.

Example

The following example opens a file and returns how many lines and sentences it contains. A sentence is defined as being any passage of text that ends in a period, a question mark, or an exclamation point.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(char *message)
{
      fprintf(stderr, "%s0, message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      FILE *fp;
      int ch, nlines, nsents;
      nlines = nsents = 0;

      /* Check number of arguments */
      if (argc != 2)
            fatal("Usage: example filename");

      /* Open file to be read */
      if ((fp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open file for reading");

      else {
            /* read lines of text */
            while ((ch = fgetc(fp)) != EOF) {
                  /* increment line count */
                  if (ch == '\n') ++nlines;
```

```
                else if (ch == '.' || ch == '!' || ch == '?') {
                      /* check if period is an ellipsis */
                      if ((ch = fgetc(fp)) != '.') {
                            /* if not, bump sentence count */
                            ++nsents;
                            /* return extra char to stream */
                            ungetc(ch, fp);
                      }

                      /* skip ellipsis */
                      else for(ch='.'; (ch=fgetc(fp))=='.';)
                            ;
                }
          }

          printf("%d line(s), %d sentence(s).\n", nlines, nsents);
    }
    return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.11
The C Programming Language, ed. 2, p. 247

See Also

fgetc, getc, getchar, scanf, STDIO

Notes

How **ungetc** affects the file-position indicator will vary, depending upon whether *fp* was opened into text mode or binary mode. If *fp* was opened into binary mode, then its file-position indicator is decremented with every successful call to **ungetc**. If, however, it was opened into text mode, then the value of the file-position indicator after a successful call to **ungetc** is unspecified; the Standard specifies only that when a character is pushed back and then re-read, the file position indicator has same value as it did when the character was first read.

union — Type

A **union** is a data type whose members occupy the same region of storage. It is used when one value may be used in a number of different circumstances. This is in contrast with a **struct**, which is a set of data elements that are laid adjacent to each other. Each object within a **union** may have its own name and distinct type.

Any object type may be contained within a **union**, including a bit-field. No incomplete object may be used. Thus, a **union** may not contain a copy of itself, but it may contain a pointer to itself. A **union** is regarded is incomplete until its closing '}' is read.

The size of a **union** is that of its largest member. Thus, a pointer to a **union** can, if correctly cast, be used as a pointer to each of the **union**'s members.

In effect, a **union** is a multiple declaration of a variable. For example, a **union** may be declared to consist of an **int**, a **double**, and a **char** *. Any one of these three elements can be held by the **union** at a time, and will be handled appropriately by it. For example, the declaration

```
union {
      int number;
      double bignumber;
      char *stringptr;
} EXAMPLE;
```

allows **EXAMPLE** to hold either an **int**, a **double**, or a pointer to a **char**, whichever is needed at the time. The elements of a **union** are accessed like those of a **struct**: for example, to access **number** from the above example, type **EXAMPLE.number**.

unions are helpful in dealing with heterogeneous data, especially within structures. However, you must keep track of what data type the **union** is holding at any given time. Assigning to a **double** within a **union** and then reading the **union** as though it held an **int** will yield results that are defined by the implementation.

A **union** initializer may only initialize the *first* member of the **union**.

Example

The following example uses a **union** to demonstrate the byte ordering of the machine upon which the program is run. It assumes that an **int** is two bytes long, and a **long** is four bytes long.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      union {
            char bytes[4];
            int words[2];
            long longs;
      } u;
      u.l = 0x12345678L;

      printf("%x %x %x %x\n",
            u.bytes[0], u.bytes[1], u.bytes[2], u.bytes[3]);
      printf("%x %x\n", u.words[0], u.words[1]);
      printf("%lx\n", u.longs);
      return EXIT_SUCCESS;
}
```

Cross-references

Standard, §3.1.2.5, §3.5.2.1
The C Programming Language, ed. 2, pp. 212*ff*

See Also

bit-field, member name, struct, tag, types

Notes

Oftentimes, **union** will be a member of a structure, and the preceding structure member will be a "tag" field, whose value indicates the type of object the **union** currently has stored. Though such a tag is required in some languages (such as Pascal), it is not required in C.

universal coordinated time — Definition

Universal coordinated time (*universel temps coordonne*, or UTC) is a universal standard of time that is based on study of an atomic clock, as corrected by comparison with pulsars. It is, for all practical purposes, identical to Greenwich Mean Time, which is the mean solar time recorded at the Greenwich Observatory in England, where by international convention the Earth's zero meridian is fixed.

Standard local time is usually calculated as an offset of UTC. For example, the time zone for Chicago is six hours (360 minutes) behind UTC, so the standard time for Chicago is calculated by subtracting 360 minutes from UTC. Calculating local time may not always be so easy, however. For example, some Islamic countries calculate local time by dividing the time between sunrise and sunset into 12 hours.

LEXICON

The function **gmtime** returns a pointer to the structure **tm** that has been initialized to hold the current UTC. The name of this function reflects the older practice of referring to Greenwich Mean Time instead of UTC.

Cross-reference

Standard, §4.12.1

See Also

broken-down time, calendar time, date and time, gmtime, local time, localtime

unlink() — Extended function (libc)

Remove a file
short unlink(char *_name_**);**

unlink removes the directory entry for the given file *name*, which in effect erases *name* from the disk. *name* cannot be open when **unlink**ed. The name is an historical artifact.

unlink returns -1 if it cannot remove a file, and zero if it can.

Example

This example removes the files named on the command line.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

/* prototype for extended function */
extern short unlink(char *name);

main(int argc, char *argv[])
{
      int i;

      for (i = 1; i < argc; i++)
      {
            if (unlink(argv[i]) == -1)
            {
                  printf("Cannot unlink \"%s\"\n", argv[i]);
                  exit(EXIT_FAILURE);
            }
      }
      exit(EXIT_SUCCESS);
}
```

See Also

extended miscellaneous, remove

Notes

unlink is not described in the ANSI Standard. Programs that use it do not strictly conform to the ANSI Standard, and may not be portable to other compilers or other environments.

The ANSI function **remove** also removes files. It is recommended that you use it instead of **unlink** so that your programs will conform more strictly to the Standard.

unsigned — C keyword

When a declaration includes the modifier **unsigned**, it indicates that the type can hold only a non-negative value.

There are four **unsigned** data types: **unsigned char**, **unsigned int**, **unsigned long int**, and **unsigned short int**. If the modifier **unsigned** is not used, the translator assumes that **int**, **long int**, and **short int** are signed. The implementation defines whether **char** is signed or unsigned by default.

An unsigned data type takes the same amount of storage as the corresponding signed type, and has the same alignment requirements.

Any value that can be represented by both a signed and an unsigned type will be represented the same way in both. An unsigned type, however, cannot represent a negative value. The sign bit is freed to hold a value. In this instance, an unsigned type can store a value of twice what can be stored in its signed counterpart.

Arithmetic that involves unsigned types will never overflow. If an arithmetic operation produces a value that is too large to fit into a particular unsigned type, that value is divided by one plus the largest value that can be held in that unsigned type, and the remainder is then stored in the unsigned type.

For information about converting one type of integer to another, see **integral types**.

When **unsigned** is used by itself, it is regarded as a synonym for **unsigned int**.

Cross-references

Standard, §3.1.2.5
The C Programming Language, ed. 2, p. 211

See Also

char, signed, types, unsigned

unsigned char — Type

An **unsigned char** is an unsigned integral type. It takes the same amount of storage as a **char**, and has the same alignment requirements.

An **unsigned char** has the minimum value of zero, and a maximum value of **UCHAR_MAX**. The last is a macro that is defined in the header **limits.h**. It is 255.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 44

See Also

char, signed char, types, unsigned

unsigned int — Type

An **unsigned int** is an unsigned integral type. It requires the same amount of storage as a **int** and has the same alignment requirements.

An **unsigned int** has the minimum value of zero, and a maximum value of **UINT_MAX**. The last is a macro that is defined in the header **limits.h**. It is 65,535.

The type **unsigned** is a synonym for **unsigned int**.

LEXICON

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

int, types, unsigned

unsigned long int — Type

An **unsigned long int** is an unsigned integral type. It requires the same amount of storage as a **long int**, and has the same alignment requirements.

An **unsigned long int** has the minimum value of zero, and a maximum value **ULONG_MAX**. The last is a macro that is defined in the header **limits.h**. It is 4,294,967,295.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

long int, types, unsigned

unsigned short int — Type

An **unsigned short int** is an unsigned integral type. It requires the same amount of storage as a **short int**, and has the same alignment requirements.

An **unsigned short int** has the minimum value of zero, and a maximum value of **USHRT_MAX**. The last is a macro that is defined in the header **limits.h**. It is 65,535.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

short int, types, unsigned

V

va_arg() — Variable arguments (stdarg.h)

Return pointer to next argument in argument list
#include <stdarg.h>
typename ***va_arg(va_list** *listptr*, *typename*);**

va_arg returns a pointer to the next argument in an argument list. It can be used with functions that take a variable number of arguments, such as **printf** or **scanf**, to help write such functions portably. It is always used with **va_end** and **va_start** within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is an object defined in the header **stdarg.h**. It must first be initialized by the macro **va_start**.

typename is the name of the type for which **va_arg** is to return a pointer. For example, if you wish **va_arg** to return a pointer to an integer, *typename* should be of type **int**.

va_arg can only handle "standard" data types, i.e., those data types that can be transformed to pointers by appending an asterisk '*'.

Example

For an example of this macro, see the entry for **variable arguments**.

Cross-references

Standard, §4.8.1.2
The C Programming Language, ed. 2, p. 254

See Also

va_end, va_start, variable arguments

Notes

If there is no next argument for **va_arg** to handle, or if *typename* is incorrect, then the behavior of **va_arg** is undefined.

va_arg must be implemented only as a macro. If its macro definition is suppressed within a program, the behavior is undefined.

va_end() — Variable arguments (libc)

Tidy up after traversal of argument list
#include <stdarg.h>
void va_end(va_list *listptr*);**

va_end helps to tidy up a function after it has traversed the argument list for a function that takes a variable number of arguments. It can be used with functions that take a variable number of arguments, such as **printf** or **scanf**, to help write such functions portably. It should be used with the routines **va_arg** and **va_start** from within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is declared in header **stdarg.h**. *listptr* must first have been initialized by macro **va_start**.

The manner of "tidying up" that **va_end** performs will vary from one computing environment to another. In many computing environments, **va_end** is not needed, and it may be implemented as an empty function.

Example

For an example of this function, see the entry for **variable arguments**.

Cross-references

Standard, §4.8.1.3
The C Programming Language, ed. 2, p. 254

See Also

va_arg, va_start, variable arguments

Notes

If **va_list** is not initialized by **va_start**, or if **va_end** is not called before a function with variable arguments exits, then behavior is undefined.

va_list — Type

Type used to handle argument lists of variable length

va_list is a **typedef** declared in the header **stdarg.h**.

va_list is used to help implement functions like **printf** and **scanf**, which can take an indeterminate number of arguments.

Example

For an example of this type, see the entry for **variable arguments**.

Cross-references

Standard, §4.8

See Also

va_arg, va_end, va_start, variable arguments

va_start() — Variable arguments (stdarg.h)

Point to beginning of argument list
#include <stdargs.h>
void va_start(va_list *listptr, type rightparm*);

va_start is a macro that points to the beginning of a list of arguments. It can be used with functions that take a variable number of arguments, such as **printf** or **scanf**, to help implement them portably. It is always used with **va_arg** and **va_end** from within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is a type defined in the header **stdarg.h**.

rightparm is the rightmost parameter defined in the function's parameter list — that is, the last parameter defined before the **...** punctuator. Its type is set by the function that is using **va_start**. Undefined behavior results if any of the following conditions apply to **rightparm**: if it has storage class **register**; if it has a function type or an array type; or if its type is not compatible with the type that results from the default argument promotions.

Example

For an example of this macro, see the entry for **variable arguments**.

Cross-references

Standard, §4.8.1.1
The C Programming Language, ed. 2, p. 254

See Also

va_arg, va_end, va_list, variable arguments

Notes

va_start must be implemented only as a macro. If the macro definition of **va_start** is suppressed within a program, the behavior is undefined.

value preserving — Definition

With respect to integral promotions, the Standard has adopted *value-preserving rules*. This may quietly break some existing code that depended on unsigned-preserving rules, as many UNIX implementations have done.

In most cases, there will be no difference in the results produced by unsigned-preserving rules and those produced by value-preserving rules. There are, however, several instances in which different results will be seen. For example:

```
long l;
unsigned int ui;
   . . .
l = ui + l;
```

In this operation, before the addition is performed, **ui** will first be promoted to type **long** if a **long** can hold the value contained in the **unsigned int**. The operation will then be performed as long addition, assigning the result to the variable **l**.

If a **long** is not large enough to represent the value contained in **ui**, which may occur under an implementation where **int**s and **long**s are the same size, then *both* **ui** and **l** are first converted to **unsigned long** before the addition is performed. Because conversion is needed to preserve the value (as opposed to the sign) of the operand as well as the result, the term "value preserving" is appropriate.

As usual, code may have to be generated to perform the conversion, and a high-quality implementation will usually issue a diagnostic message in such a case.

Cross-references

Standard, §3.2
The C Programming Language, ed. 2, pp

See Also

conversions, integral promotions

variable arguments — Overview

The Standard mandates the creation of a set of routines to help implement functions, such as **printf** and **scanf**, that take a variable number of arguments. These routines are called from within another function to help it handle its arguments. If the ellipsis punctuator '...' appears at the end of the list of arguments in a function's prototype, then that a function can take a variable number of arguments.

These routines are declared or defined in the header **stdarg.h**, and are as follows:

va_arg	Return pointer to next argument in argument list
va_end	Tidy up after an argument list has been traversed
va_start	Initialize object that holds function arguments

va_arg and **va_start** must be implemented as macros; **va_end** must be implemented as a library function. All three use the special type **va_list**, which is an object that holds the arguments to the function being implemented.

LEXICON

Example

The following example concatenates multiple strings into a common allocated string and returns the string's address.

```
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>

char *
multcat(int numargs, ... )
{
      va_list argptr;
      char *result;
      int i, siz;

      /* get size required */
      va_start(argptr, numargs);
      for(siz = i = 0; i < numargs; i++)
            siz += strlen(va_arg(argptr, char *));

      if ((result = calloc(siz + 1, 1)) == NULL) {
            fprintf(stderr, "Out of space\n");
            exit(EXIT_FAILURE);
      }
      va_end(argptr);

      va_start(argptr, numargs);
      for(i = 0; i < numargs; i++)
            strcat(result, va_arg(argptr, char *));
      va_end(argptr);
      return(result);
}

int
main(void)
{
      printf(multcat(5, "One ", "two ", "three ",
            "testing", ".\n"));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.8
The C Programming Language, ed. 2, p. 254

See Also

Library, stdarg.h, va_list

vfprintf() — STDIO (libc)

Print formatted text into stream
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE*fp***, const char** **format***format***, va_list** *arguments***);**

vfprintf constructs a formatted string and writes it into the stream pointed to by *fp*. It translates integers, floating-point numbers, and strings into a variety of text formats. **vfprintf** can handle a variable list of arguments of various types. It is roughly equivalent to the 'r' conversion specifier to **fprintf**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how particular data type is converted into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence "%%".) See **printf** for further discussion of the conversion specification, and for a table of the type specifiers that can be used with **vfprintf**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header **stdarg.h**. It has been initialized by the macro **va_start** and points to the base of the list of arguments used by **vfprintf**. For more information, see **variable arguments**. *arguments* should access one argument for each conversion specification in *format*, of the type appropriate to its conversion specification.

For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char** *. *arguments* can take only the data types acceptable to the macro **va_arg**; namely, basic types that can be converted to pointers simply by adding a '*' after the type name. See **va_arg** for more information on this point.

If there are fewer arguments than conversion specifications, then **vfprintf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding conversion specifier, then the behavior of **vfprintf** is undefined. Thus, presenting an **int** where **vfprintf** expects a **char** * may generate unwelcome results.

If it writes the formatted string correctly, **vfprintf** returns the number of characters written. Otherwise, it returns a negative number.

Example

This example sets up a standard multiargument error message. It is the source of the function **fatal**, which is used throughout this manual.

```
#include <math.h>
#include <stdarg.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(char *format, ...)
{
      va_list argptr;

      /* if there is a system message, display it */
      if(errno)
            perror(NULL);

      /* if there is a user message, use it */
      if(format != NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}
```

LEXICON

```
main(void)
{
      /*
       * This is guaranteed to be wrong.  It should push
       * an error code into errno.
       */
      sqrt(-1.0);

      /* Now, show the messages */
      fatal("A %s error message%c", "complex", '\n');

      /* If we get this far, something is wrong */
      return(EXIT_FAILURE);
}
```

Cross-references

Standard, §4.9.6.7
The C Programming Language, ed. 2, p. 245

See Also

fprintf, printf, sprintf, STDIO, vprintf, vsprintf

Notes

vfprintf can construct a string up to at least 509 characters long.

The character that **vfprintf** uses to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

void — C keyword

Empty type

The term **void** indicates the empty type. The following sections describe the ways it is used.

Function Type

void can be used in a function prototype or definition to indicate that a function returns no value. For example, the declaration

```
void example();
```

indicates that the function **example** returns nothing. It would be an error for **example** to attempt to return a value to a function that calls it, or for the calling function to use its value in an expression.

Function Arguments

void can also be used in a function prototype or function declaration to indicate that a function has no arguments. For example, the declaration

```
void example(void);
```

indicates that the function **example** not only returns nothing, but it takes no arguments as well. The older practice of writing **example()** remains legal. But as before, it indicates merely that nothing is said about arguments.

Void Expression

void can be used to indicate that the value of an expression is to be ignored. This is done by casting the expression to type **void**. Prefacing an expression with the cast **(void)** throws away its value (i.e., casts it into the void), although the expression is evaluated for possible side-effects.

void *

A **void** * ("pointer to void") is a generic pointer. It is used in much the same way that **char** * ("pointer to **char**") was used in earlier descriptions of C. The new generic pointer type eliminates the earlier confusion between a pointer to **char** (e.g., a string pointer) and a generic pointer.

Because by definition the **void** type includes no objects, a pointer to **void** may not be dereferenced. That is, you should not directly access the object to which it points by using the indirection operator '*'. In the code

```
void *voidp;
   . . .
if (*voidp > 0)
   . . .
```

the behavior of dereferencing the pointer to **void** is undefined. It may or may not generate an error; if it does not, the results may be unpredictable.

It is correct, however, to cast a pointer to **void** to a standard object pointer type and then dereference it. For example, the code

```
void *voidp;
   . . .
if (*(char *)voidp > 0)
   . . .
```

is permitted.

The Standard guarantees that a pointer to **void** may be converted to a pointer to any incomplete type or object type. It also guarantees that a pointer to any incomplete type or object type may be converted into a pointer to **void**. Moreover, converting the result back to the original type results in a pointer equal to the original pointer. That is, conversion of any object pointer type to **void** * and back again does not change the representation of the pointer. However, if an object pointer is converted to **void** * and then converted to a pointer to a type whose alignment is stricter than that of the original type, behavior is undefined.

The Standard also guarantees that the pointer types **char** * and **void** * have the same representation. This prevents the Standard from breaking existing code for functions with generic-pointer arguments (previously defined using type **char** * but now defined with type **void** *).

The introduction of the generic pointer **void** * by the Standard serves several purposes in addition to those noted above. The Standard no longer allows comparison between pointers of different types, except that any object pointer may be compared to a **void** *. Casting object pointers with the expression

```
(void *)
```

allows comparisons that would otherwise be illegal. Library functions that have commonly been written with pointers of various types as arguments (such as **fread**) can be defined with a prototype **void** * argument, which allows the arguments to be quietly converted to the correct type.

The generic pointer **void** * is also used as the type of the value returned by some functions (e.g., **malloc**), to indicate that the returned value is a pointer to something of indeterminate type.

Cross-references

Standard, §3.1.2.5, §3.2.2.2-3, §3.3.4, §3.5.2, §3.5.3.1, §3.5.4.3
The C Programming Language, ed. 2, pp. 199, 218

See Also

NULL, pointer, precedence, types

LEXICON

void expression — Definition

A *void expression* is any expression that has type **void**. By definition, it has no value; therefore, its value cannot be assigned to any other expression. Normally, a void expression is used for its side-effects.

If an expression of any other type is used in a situation that requires a void expression, the value of that expression is discarded.

Cross-reference

Standard, §3.2.2.2

See Also

conversions

volatile — C keyword

Qualify an identifier as frequently changing

The type qualifier **volatile** marks an identifier as being frequently changed, either by other portions of the program, by the hardware, by other programs in the execution environment, or by any combination of these. This alerts the translator to re-fetch the given identifier whenever it encounters an expression that includes the identifier. In addition, an object marked as **volatile** must be stored at the point where an assignment to this object takes place.

Cross-references

Standard, §3.5.3
The C Programming Language, ed. 2, p. 211

See Also

const, type qualifier

Notes

volatile was created by the Committee for systems' programs that deal with memory-mapped I/O or ports where the program is not the only task that may modify the given port in memory. **volatile** tells the translator that it does not know everything that is happening to the object.

Another use for **volatile** is for translators that perform optimizations, such as deferring storage of registers or peephole optimization. **volatile** requires that the object be read and stored at exactly those points where the program has specified these actions.

vprintf() — STDIO (libc)

Print formatted text into standard output stream
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char ******format***, va_list** *arguments***);**

vprintf constructs a formatted string and writes it into the standard output stream. It translates integers, floating-point numbers, and strings into a variety of text formats. **vprintf** can handle a variable list of arguments of various types. It is roughly equivalent to the 'r' conversion specifier to **printf**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification defines how a particular data type is converted into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence "%%".) See **printf** for further discussion of the conversion specification and for a table of the type specifiers that can be used with **vprintf**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header **stdarg.h**. It has been initialized by the macro **va_start** and points to the base of the list of arguments used by **vprintf**. For more information, see **variable arguments**. *arguments* should access one argument for each conversion specification in *format* of the type appropriate to conversion specification.

For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char** *.

If there are fewer arguments than conversion specifications, then **vprintf**'s behavior is undefined. If there are more, every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding type specification, then the behavior of **vprintf** is undefined; thus, accessing an **int** where **vprintf** expects a **char** * may generate unwelcome results.

If it writes the formatted string correctly, **vprintf** returns the number of characters written; otherwise, it returns a negative number.

Cross-references

Standard, §4.9.6.8
The C Programming Language, ed. 2, p. 245

See Also

fprintf, printf, sprintf, STDIO, vfprintf, vsprintf

Notes

vprintf can construct a string up to at least 509 characters long.

The character that **vprintf** uses to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

Each *argument* must have basic type, which can be converted to a pointer simply by adding an '*' after the type name. This is the same restriction that applies to the arguments to the macro **va_arg**.

vsprintf() — STDIO (libc)

Print formatted text into string
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char ******string***, const char** ******format***, va_list** *arguments***);**

vsprintf constructs a formatted string in the area pointed to by *string*. It translates integers, floating-point numbers, and strings into a variety of text formats. **vsprintf** can handle a variable list of arguments of various types. It is roughly equivalent to the 'r' conversion specifier to **sprintf**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how to convert a particular data type into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence "%%".) See **printf** for further discussion of the conversion specification and for a table of the type specifiers that can be used with **vsprintf**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header **stdarg.h**. It has been initialized by the macro **va_start** and points to the base of the list of arguments used by **vsprintf**. For more information, see **variable arguments**. *arguments* should access one argument for each conversion specification in *format* of the type appropriate to the conversion specification.

For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char** *.

LEXICON

If there are fewer arguments than conversion specifications, then **vsprintf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding type specification, then the behavior of **vsprintf** is undefined; thus, accessing an **int** where **vsprintf** expects a **char** * may generate unwelcome results.

If it writes the formatted string correctly, **vsprintf** returns the number of characters written; otherwise, it returns a negative number.

Cross-references

Standard, §4.9.6.7
The C Programming Language, ed. 2, p. 245

See Also

fprintf, printf, sprintf, STDIO, vprintf, vsprintf

Notes

vsprintf can construct a string up to at least 509 characters long.

The character that **vsprintf** uses to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

W

wc — Command

Count words, lines, and characters in files
wc [-clw] [*file…***]**

wc counts words, lines, and characters in each *file* named. If no *file* is given, **wc** uses the standard input. If more than one *file* is given, **wc** also prints a total for all of the files.

A *word* is a string of characters surrounded by white space (blanks, tabs, or newlines).

Options control the printing of various counts:

-c Print a count of character.

-l Print a count of lines.

-w Print a count of words.

The default action is to print all counts.

See Also

commands

wcstombs() — General utility (libc)

Convert sequence of wide characters to multibyte characters
#include <stdlib.h>
size_t wcstombs(wchar_t ******multibyte***, const char** ******widechar***, size_t** *number***);**

The function **wcstombs** converts a sequence of wide characters to their corresponding multibyte characters. It is the same as a series of calls of the type:

```
wctomb(multibyte, *widechar);
```

except that the call to **wcstombs** does not affect the internal state of **wctomb**.

widechars points to the base of the sequence of wide characters to be converted to multibyte characters. *multibyte* points to the area into which the characters will be written. The sequence begins and ends in an initial shift state. *number* is the number of characters to be converted. **wcstombs** converts characters either until it reads and converts the null character that ends the sequence, or until it has converted *number* characters. In the latter case, no null character is written at the end of the sequence of multibyte characters.

wcstombs returns -1 cast to **size_t** if it encounters an invalid wide character before it has converted *number* characters. Otherwise, it returns the number of characters converted, excluding the null character that ends the sequence.

Cross-reference

Standard, §4.10.7.4

See Also

general utilities, mbstowcs

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

LEXICON

wctomb() — General utility (libc)

Convert a wide character to a multibyte character
#include <stdlib.h>
int wctomb(char **string*, **wchar_t** *widecharacter*);

wctomb converts *widecharacter* to its corresponding multibyte character and stores the result in the area pointed to by *string*.

If *string* is set to NULL, then **wctomb** merely checks to see if the current set of multibyte characters include state-dependent encodings. It returns zero if the set does not include state-dependent codings, and a number other than zero if it does.

If *string* is set to a value other than NULL, then **wctomb** does the following:

1. It returns zero if *widecharacter* is zero.
2. It returns -1 if the value of *widecharacter* does not correspond to a legitimate multibyte character for the present locale.
3. If the value of *widecharacter* does correspond to a legitimate multibyte character, then it returns the number of bytes that comprise that character.

wctomb never returns a value greater than that of the macro **MB_CUR_MAX**.

Cross-reference

Standard, §4.10.7.5

See Also

general utilities, MB_CUR_MAX, mblen, mbtowc, wchar_t

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

The address pointed to by *string* should have **MB_CUR_MAX** bytes of storage allocated to it. If not, you may overwrite memory currently in use.

while — C keyword

Loop construct
while(*condition***)** *statement*

while introduces a conditional loop. Unlike a **do** loop, a **while** loop tests *condition* before execution of *statement*. The loop ends when *condition* is no longer satisfied. Hence, the loop may not execute at all, if *condition* is initially false.

For example,

```
while (variable < 10)
```

introduces a loop whose statements will continue to execute until **variable** is equivalent to ten or greater. The statement

```
while (1)
```

will loop until interrupted by **break**, **goto**, or **return**.

Example

For an example of this statement, see **sscanf**.

Cross-references

Standard, §3.6.5.1
The C Programming Language, ed. 2, pp. 60*ff*

See Also

C keywords, do, for, statements, while

wildcards — Definition

Wildcards are characters that, under special circumstances, represent a range of ASCII characters. Another name for them is "metacharacters". The wildcards available under MS-DOS are as follows:

? Match any one character.

***** Match any number of characters, or no characters at all.

See Also

Definitions, egrep, patterns, pnmatch

write() — Extended function (libc)

Write into a file
short write(short *fd***, char** ******buffer***, short** *n***);**

write writes *n* bytes of data, beginning at address *buffer*, into the file *fd*. Writing begins at the current write position, as set by the last call to either **write** or **lseek**. **write** advances the position of the file pointer by the number of characters written.

write returns -1 if an error occurred before the **write** operation commenced, such as if *fd* is bad or *buffer* contains an invalid address. Otherwise, it returns the number of bytes actually written. It should be considered an error if this number is not the same as *n*.

Example

For an example of how to use this function, see the entry for **open**.

See Also

extended miscellaneous

Notes

write is a low-level call that passes data directly to MS-DOS. It should not be intermixed with high-level calls, such as **fread**, **fwrite**, or **fopen** without care.

write is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or to other environments.

X

xctype.h — Header

#include <xctype.h>

In addition to the character-handling functions described in the Standard, **Let's C** includes the following extended character-handling functions and macros:

_tolower	Change a character to lower case
_toupper	Change a character to upper case
isascii	See if a character is in the ASCII character set
toascii	Convert a character to printable ASCII

These functions and macros are declared or defined in the header **xctype.h**. In previous releases of **Let's C**, they had been declared in the header **ctype.h**. This change was made to conform to the Standard, and may require that some code be altered.

A program that uses any of these routines no longer conforms strictly to the Standard, and may not be portable to other compilers or environments.

See Also

ctype.h, extended character handling, header

XOFF — Manifest constant

XOFF is a flow-control signal used with asynchronous communications. Usually, it consists of a **<ctrl-S>** character (octal 023). It is sent by the receiving device when its asynchronous buffer is nearly full, or has reached the "high-water mark".

When XOFF is used to help control data transmission, binary files cannot be transmitted.

See Also

ASCII, Environment, XON

XON — Manifest constant

XON is a flow-control signal used with asynchronous communications. Usually, it consists of a **<ctrl-Q>** character (octal 021). It is sent by the receiving device when its asynchronous buffer is nearly empty, or has reached the "low-water mark".

When XON is used to help control data transmission, binary files cannot be transmitted.

See Also

ASCII, Environment, XOFF

xtime.h — Header

#include <xtime.h>

xtime.h is a header that holds prototypes for the extended time functions included with **Let's C**:

Time conversion

timezone	Seconds from UTC to give local time
dayspermonth	How many days in this historical month?
dstadjust	Seconds to local standard, if any
isleapyear	Is this year AD a leap year?
tzname	Array with names of standard and daylight times

Julian time

time_to_jday	Convert **time_t** to the Julian date
jday_to_time	Convert Julian date to **time_t**

tm_to_jday	Convert **tm** structure to Julian date
jday_to_tm	Convert Julian date to **tm** structure

xtime.h also declares the structure **jday**.

See Also

extended time, header, time.h

Notes

To conform to the ANSI Standard, these functions were moved from the header **time.h** to the header **xtime.h**. This may require that some code be altered.

Appendix

The following lists all of the entries in the Lexicon in their logical order. The Lexicon is tree structured, with the root entry being the one entitled **Lexicon**.

The logical structure of the Lexicon closely follows that of the ANSI Standard. Articles on related topics are grouped together for easy access. In instances where an article describes an entity that has more than one use (e.g., the operator '*'), the article's position in the logic tree is based on a judgement of how that entity is used most frequently by C programmers.

Each entry marked with an asterisk '*' refers to a topic that is specific to the Atari ST or to **Let's C**.

Lexicon
- Definitions
  - address
  - alias
  - alignment
  - argument
  - arena*
  - ASCII
  - behavior
  - BIOS*
  - bit
  - bit-fields
  - bit map*
  - block
  - buffer
  - byte
  - compliance
  - cc0*
  - cc1*
  - cc2*
  - cc3*
  - cc4*
  - daemon*
  - decimal-point character
  - directory*
  - domain error
  - executable file*
  - false
  - field*
  - file
  - file descriptor*
  - interrupt*
  - letter
  - link
  - manifest constant
  - nested comments*
  - nybble*
  - object format*
  - object

Appendix

- program startup
- program termination
- runtime startup*
  - _main*
  - crts0xl*
  - crts0xs*
- sequence points
- side effects
- signals/interrupts
- source file
- translation limits
- translation phase
- translation unit
- trigraph sequences

Language

- constant expressions
- conversions
  - explicit conversion
  - function designator
  - implicit conversion
  - lvalue
  - null pointer constant
  - value preserving
  - void expression
- declarations
  - definition
  - declarators
    - array declarators
    - function declarators
    - pointer declarators
  - initialization
  - storage-class specifiers
    - auto
    - extern
    - register
    - static
    - typedef
  - type names
  - type qualifier
    - const
    - volatile
  - type specifier
    - compatible types
    - enum
- expressions
  - !
  - ! =
  - %
  - % =
  - &
  - & &
  - & =
  - *
  - * =

+
+ +
+ =
,
-
- -
- =
->
.
/
/ =
<
< <
< < =
< =
=
= =
>
> >
> > =
> =
? :
[ ]
^
^ =
|
| |
| =
~
sizeof
external definitions
function definition
object definition
function call
function prototypes
lexical elements
comment
*/
/*
constants
character constant
escape sequence
enumeration constant
floating constant
integer constant
header names
identifiers
digit
external name
internal name
linkage
nondigit
name space
label

member
ordinary identifier
tag
scope
storage duration
types
char
double
float
int
long double
long int
pointer
short int
signed
signed char
struct
union
unsigned
unsigned char
unsigned int
unsigned long int
unsigned short int
void
keywords
operators
preprocessing numbers
punctuators
( )
:
;
{ }
string literal
"
token
preprocessing
#
# #
#define
#elif
#else
#endif
#error
#if
#ifdef
#ifndef
#include
#line
#pragma
#undef
_ _DATE_ _
_ _FILE_ _
_ _LINE_ _
_ _STDC_ _

- _ _TIME_ _
- defined
- null directive

statements
- alien*
- break
- case
- continue
- default
- do
- else
- for
- goto
- if
- null statement
- return
- switch
- while

Library
- libcxl*
- libcxl87*
- libcxs*
- libcxs87*
- libmxl*
- libmxl87*
- libmxs*
- libmxs87*
- NULL
- offsetof()
- ptrdiff_t
- size_t
- stddef.h
- wchar_t
- access checking*
  - access.h*
  - access()*
  - path.h*
  - path()*
  - stat.h*
  - stat()*
- character handling
  - ctype.h
  - isalnum()
  - isalpha()
  - iscntrl()
  - isdigit()
  - isgraph()
  - islower()
  - isprint()
  - ispunct()
  - isspace()
  - isupper()
  - isxdigit()
  - tolower()

- toupper()
- date and time
  - time.h
  - CLK_TCK
  - clock_t
  - time_t
  - tm
  - asctime()
  - clock()
  - ctime()
  - difftime()
  - gmtime()
  - localtime()
  - mktime()
  - strftime()
  - time()
- diagnostics
  - assert.h
  - assert()
- errors
  - errno.h
  - errno
- extended character handling
  - xctype.h*
  - _tolower()*
  - _toupper()*
  - isascii()*
  - toascii()*
- extended mathematics
  - xmath.h
  - cabs()*
  - hypot()*
  - j0()*
  - j1()*
  - jn()*
- extended miscellaneous
  - _exit()*
  - _zero()*
  - close()*
  - creat()*
  - dup()*
  - dup2()*
  - ecvt()*
  - exargs()*
  - execall()*
  - fcvt()*
  - gcvt()*
  - in()*
  - inb()*
  - index()*
  - mktemp()*
  - notmem()*
  - open()*
  - out()*

- outb()*
- peek()*
- peekb()*
- pnmatch()*
- poke()*
- pokeb()*
- read()*
- rindex()*
- sbrk()*
- shellsort()*
- swab()*
- tempnam()*
- unlink()*
- write()*

extended STDIO
- xstdio.h
- fdopen()*
- fgetw()*
- fileno()*
- fputw()*
- getanb()*
- getcnb()*
- getw()*
- lseek()*
- putanb()*
- putcnb()*
- putw()*

extended time
- xtime.h*
- dayspermonth()*
- isleapyear()*
- jday_to_time()*
- jday_to_tm()*
- stime()*
- time_to_jday()*
- tm_to_jday()*

general utilities
- stdlib.h
- div_t
- ldiv_t
- abort()
- abs()
- atexit()
- atof()
- atoi()
- atol()
- bsearch()
- calloc()
- div()
- exit()
- free()
- getenv()
- labs()
- ldiv()

Appendix

- malloc()
- mblen()
- mbstowcs()
- mbtowc()
- qsort()
- rand()
- realloc()
- srand()
- strtod()
- strtol()
- strtoul()
- system()
- wcstombs()
- wctomb()

header
- bios.h*
- larges.h*
- mtype.h*

i8086 support*
- dos.h*
- PTR*
- _copy()*
- csreg()*
- dsreg()*
- esreg()*
- intcall()*
- ptoreg()*
- regtop()*
- ssreg()*

localization
- locale.h
- lconv
- localeconv()
- setlocale()

mathematics
- math.h
- acos()
- asin()
- atan()
- atan2()
- ceil()
- cos()
- cosh()
- exp()
- fabs()
- floor()
- fmod()
- frexp()
- ldexp()
- log()
- log10()
- modf()
- pow()
- sin()

- sinh()
- sqrt()
- tan()
- tanh()

non-local jumps
- setjmp.h
- jmp_buf
- longjmp()
- setjmp()

signal handling
- signal.h
- sig_atomic_t
- raise()
- signal()

STDIO
- stdio.h
- EOF
- FILE
- fpos_t
- stderr
- stdin
- stdout
- clearerr()
- fclose()
- feof()
- ferror()
- fflush()
- fgetc()
- fgetpos()
- fgets()
- fopen()
- fprintf()
- fputc()
- fputs()
- fread()
- freopen()
- fscanf()
- fseek()
- fsetpos()
- ftell()
- fwrite()
- getc()
- getchar()
- gets()
- perror()
- printf()
- putc()
- putchar()
- puts()
- rewind()
- setbuf()
- setvbuf()
- remove()
- rename()

Appendix

- scanf()
- sprintf()
- sscanf()
- tmpfile()
- tmpnam()
- ungetc()
- vfprintf()
- vprintf()
- vsprintf()

string handling
- string.h
- memchr()
- memcpy()
- memcmp()
- memmove()
- memset()
- strcat()
- strchr()
- strcmp()
- strcoll()
- strcpy()
- strcspn()
- strerror()
- strlen()
- strncat()
- strncmp()
- strncpy()
- strpbrk()
- strrchr()
- strspn()
- strstr()
- strtok()
- strxfrm()

variable arguments
- stdarg.h
- va_list
- va_arg()
- va_end()
- va_start()

DOS-specific features

command*
- as*
- cc*
- cmp*
- cpp*
- egrep*
- exetcom*
- fixobj*
- ld*
- make*
- me*
- mwlib*
- nm*
- size*

- strip*
- tail*
- wc*

example*
- example*
- picture*

technical information*
- ansi.sys*
- BIOS data area
- byte ordering*
- i8087*
- keyboard
- LARGE model
- model
- SMALL model

Index

 # to _

A

B

C

F

G

H

M

INDEX

P

INDEX

T

INDEX

X

Y

User Reaction Report

To keep this manual free of errors and to facilitate future improvements, we will appreciate receiving your reactions. Please fill in the appropriate sections below, detach, and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name:

Company:

Address:

Phone:

Date:

Version and hardware used:

Did you find any errors in the manual?

Can you suggest any improvements to the manual?

Did you find any bugs in the software?

Can you suggest improvements or enhancements to the software?

Additional comments:

OTHER QUALITY PRODUCTS FROM

THE MARK WILLIAMS COMPANY

CSD C SOURCE DEBUGGER **$75.00**

Once you've written your Let's C programs, you'll want the CSD C Source Debugger to help you get them running. CSD lets you look directly at your source code as you debug. That's a big help for novices, and a real time-saver for experienced programmers. BYTE magazine writes, "CSD is close to the ideal debugging environment...." For IBM and compatibles.

LET'S C UTILITIES **$39.95**

This collection of UNIX programming tools will increase your productivity and streamline your programming. The Let's C Utilities include *ed*, the classic line editor; *diff*, which compares two files; *m4*, a powerful macro processor; *sort*, a versatile file-sorting utility; and *uniq*, which removes repeat lines in a file. Together with Let's C and CSD, these utilities provide a complete C programming system. For IBM and compatibles.

FAST FORWARD **$69.95**

Wish your computer could run faster? With Fast Forward it can! Fast Forward is a software utility that makes all your software run up to 10 times faster. By using your computer's high-speed internal memory (RAM), Fast Forward bypasses the slow disk drive. Programs requiring frequent disk access will show amazing improvements. For IBM and compatibles.

MARK WILLIAMS C FOR THE ATARI ST **$179.95**

If you're programming in C on an Atari ST, this is the C compiler for you. It's faster than ever, and includes GEM documentation, a symbolic debugger, an integrated editor for easy bug fixes, and even a RAM disk. Analog Computing called it "the all-around best choice for serious software development on the ST." Over a hundred complete sample programs make it ideal for novices, too.

60 Day Money Back Guarantee

For more information on any of these products,
call (312) 472-6659

Order Form

You may order additional products from the Mark Williams Company with this convenient card. Simply fill out the form, then detach, fold and seal it. Postage is pre-paid, and we will ship your order the same day we receive your request.

Quantity	*Product*	*Unit Price*
____	**Let's C** (IBM PC)	$75.00
____	**csd C Source Debugger** (IBM PC)	$75.00
____	**Let's C Utilities** (IBM PC)	$39.95
____	**Fast Forward** (IBM PC)	$69.95
____	**Mark Williams C** (Atari ST)	$179.95
____	**COHERENT** (IBM PC)	$495.00
____	**XYBASIC** (z80, 8080)	Call
____	**C Cross-Compilers**	Call

If you would like information on any of these products, put a check next to the product above and then check the Information Only box below.

60 Day Money Back Guarantee

- - - - - - - - - - - - - - - - - - fold along line - - - - - - - - - - - - - - - - - -

Please clearly indicate the method of payment on all orders and provide your complete shipping address.

[ ] Check
[ ] Money Order
[ ] Information Only

[ ] Visa, Master Card, American Express
Card #:_____
Expiration Date:_____

Name _____

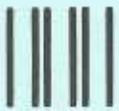
Address _____

City _____ State _____ Zip _____

Daytime Phone (_____) _____ - _____

Signature _____

Fold and seal order form before mailing.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS • PERMIT NO. 10820 • CHICAGO, IL

POSTAGE WILL BE PAID BY ADDRESSEE

Mark Williams Company
1430 Wrightwood Avenue
Chicago, Illinois 60614

MARK WILLIAMS COMPANY ("MWC")
Software License Agreement

YOU SHOULD CAREFULLY READ THIS SOFTWARE LICENSE AGREEMENT BEFORE BREAKING THE SEAL ON THE DISKETTE ENVELOPE. BREAKING THE SEAL INDICATES YOUR ACCEPTANCE OF THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE DISKETTE UNOPENED, AND YOUR MONEY WILL BE REFUNDED.

MWC provides this software and licenses its use to you. You assume responsibility for the selection of the software to achieve your intended results, and for the installation, use and results obtained from it.

LICENSE

MWC grants you a license only to: (a) use the software on a single machine; and (b) copy the software into any machine readable form for backup purposes in support of your use of the software on the single machine, unless the software includes mechanisms to limit or inhibit copying.

As an exception to the foregoing paragraph, we grant you the right to include portions of the MWC Runtime Library (as defined below) in software programs that you develop, called Composite Programs, and to use, distribute and license Composite Programs to third parties without payment of any further license fee. You shall, however, include in the object code for each Composite Program a notice in this form: "Portions of this program, copyright 1984-1987, Mark Williams Company." As an express condition to the use of the software, you agree to indemnify and hold MWC harmless from all claims by you and third parties arising out of the use of any Composite Program. Any portion of the Runtime Library merged into another program will continue to be subject to the terms and conditions of this Agreement. "Runtime Library" is defined as the set of copyrighted MWC language subroutines provided with the software, a portion of which must be linked to and become part of a Composite Program for that Program to run on a computer.

You may not use, copy or modify the software except as expressly provided for in this license. You may not transfer the software, or any copy, modification or merged portion, in whole or in part.

TERM

You may terminate the license at any time by destroying the software together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any terms or conditions of this Agreement. You agree upon termination to destroy the software together with all copies, modifications and merged portions in any form. The license shall terminate upon termination of this Agreement.

LIMITED WARRANTY

EXCEPT FOR THE LIMITED WARRANTY SET FORTH IN THE NEXT PARAGRAPH, THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT MWC OR ANY AUTHORIZED MWC DEALER) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. MWC DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR FREE.

MWC warrants to the original licensee that the diskette(s) on which the software program is recorded is free from defects in material and workmanship under normal use and service for a period of 60 days from the delivery date as evidenced by a copy of your receipt. Your exclusive remedy is replacement of the defective diskette(s) provided that you return it to MWC with a copy of your receipt.

IN NO EVENT WILL MWC BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF MWC OR AN AUTHORIZED MWC DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

MISCELLANEOUS

You may not sublicense, assign or transfer the license to the software except as expresly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void. This Agreement will be governed by the laws of the State of Illinois.

Should you have any questions concerning this Agreement, you may contact MWC by writing to Mark Williams Company, 1430 W. Wrightwood Ave., Chicago, Illinois 60614.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Let's C®

THE FAST, POWERFUL, COMPLETE COMPILER!

If you want it all–power, speed, proven performance and value–you can't do better than Let's C. Beginner or professional, this is the one compiler you need.

Let's C tears through compiling, producing the tight, fast code you're after. Makes error correction a snap with an interactive editor that automatically positions you at any problem code. Enhances the power and flexibility of C with indispensable libraries and utilities. Guides you through fundamentals, advanced concepts and dozens of usable examples with clearly organized, complete documentation.

Mark Williams compilers have stood the test of tough customers and the test of time. DEC, Intel and Wang systems are sold with the reliable, high performance C compiler from Mark Williams. A name that's been on quality programming tools since 1976.

FEATURES

Let's C Compiler

- Fast in-memory compilation
- Fast, compact code
- Large and small memory models
- Full Kernighan & Ritchie C plus ANSI extensions
- Register variables
- Integrated environment or command line interface
- 8087 sensing and support
- cc one-step compile and link
- Supported by dozens of third-party libraries
- MS-DOS object compatible
- Not copy protected

MicroEMACS Full-Screen Editor

- Automatically positions cursor at problem code during compiling
- Automatic recompile from editor
- Up to 11 windows available for editing
- Keyboard macros
- Complete commented source code included for customization

Utilities and Libraries

- Full-featured assembler
- Powerful utilities including: Make, Archiver, egrep, tail, and wc
- Complete UNIX-compatible libraries

Documentation

- "C for beginners" tutorial included
- Dozens of examples to use in your own programs
- Unique Lexicon format lets you quickly find the information you need
- Tutorials for MicroEMACS and *make* utility
- Detailed explanation of error messages

Also available for use with Let's C: *csd* C SOURCE DEBUGGER

- Lets you debug in original C source code, instead of assembler
- Large and small memory models

What the reviewers say:

"The performance and documentation of the $75 Let's C compiler rival those of C compilers for the PC currently being sold for $500."
– Marty Franz, PC TECH JOURNAL

"Let's C is an inexpensive, high-quality programming package. It comes complete with all the tools you will need to create applications…
csd *is a definite aid to learning C and an indispensable tool for program development… close to the ideal debugging environment."*
– William G. Wong, BYTE

Let's C runs on IBM PCs and 100% compatibles with MS-DOS 2.0 or greater.
Let's C requires 320K of RAM (384K recommended) and two double-sided 320K disk drives or hard disk.
Let's C is a registered trademark of the Mark Williams Company.
UNIX is a trademark of Bell Labs.

ISBN 0-927860-03-1