

TRS-80[®] C

Radio Shack[®]

A DIVISION OF TANDY CORPORATION
FORT WORTH, TEXAS 76102

THIS WARRANTY SUPERSEDES ALL PRIOR WARRANTIES

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER's sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. EXCEPT AS PROVIDED HEREIN, RADIO SHACK MAKES NO EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE."

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on one computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on one computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software only for backup or archival purposes or if additional copies are required in the operation of one computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the original CUSTOMER specific legal rights, and the original CUSTOMER may have other rights which vary from state to state.

TRS-80[®] C

Radio Shack[®]
A DIVISION OF TANDY CORPORATION
FORT WORTH, TEXAS 76102

COPYRIGHT NOTICES

Model 4 TRS-80 C Manual
Copyright 1983 by Alcor Systems
Licensed to Tandy Corporation
All rights reserved

Reproduction or use, without express written permission from Tandy Corporation and Alcor Systems of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure accuracy, Tandy Corporation and Alcor Systems assume no liability resulting from any errors or omissions in this manual or from the use of the information obtained herein.

Model 4 TRS-80 C Software
Copyright 1983 by Alcor Systems
Licensed to Tandy Corporation
All rights reserved

TRSDOS 6 Operating System
Copyright 1983 by Logical Systems
Licensed to Tandy Corporation
All rights reserved

INTRODUCTION

Congratulations on the purchase of the Model 4 TRS-80 C programming system. TRS-80 C is a complete program development system that will increase your productivity as a programmer.

TRS-80 C is a complete implementatation of C as defined in "The C Programming Language" by Kernighan and Ritchie. The C library routines provided are Unix compatible. The compatibility of TRS-80 C with other Unix implementations of C makes it easy to move C programs from other computers to the Model 4 or vice versa.

Included with the TRS-80 C programming system is a very powerful, programmable, full screen text editor. The editor characteristics may be easily changed to suit your personal preferences. You can map editor commands to the Model 4 keyboard as desired and you can define your own editor commands.

The TRS-80 C compiler generates a very efficient and compact object code. Some programs developed with TRS-80 C will execute up to 50 times faster than equivalent programs developed with interpreted BASIC, depending on the features used. The compact size of the object code allows you to develop reasonably large programs without the need to resort to overlays or chaining.

An added feature of TRS-80 C is compatibility with TRS-80 Pascal (Cat. No. 26-2211 and 26-2212). You can call functions or procedures written in TRS-80 Pascal from a TRS-80 C program or vice versa.

How TRS-80 C Works

TRS-80 C is a compiled language. This means that programs must first be translated to object format before they may be executed.

The first step in developing a program is to enter the program into the computer and save it to a disk file. A full screen, customizable text editor (EDIT/CMD) is supplied to allow you to create your programs.

The second step is to compile the program. The C compiler (CC/CMD) is a fast one pass compiler that generates object code that may be directly executed.

The third step is to execute the program. There is a run utility (RUNC/CMD) supplied which will execute your compiled programs. The run utility loads and executes object format files created by the compiler.

The linking loader utility (LINKLOAD/CMD) must be used to execute programs which have been split into separately compiled segments. The linking loader loads one or more object format files and links them into a single executable program. It has the ability to execute the program directly or to build an executable command file.

Optional optimizations may be performed to decrease the size of a compiled program or to increase its execution speed. The optimize utility (OPTIMIZE/CMD) reduces the the size of an object format file by 10 to 30 percent. The codegen utility (CODEGEN/CMD) translates an object format file into machine instructions which increases execution speed 3 to 5 times.

Producing Programs for Resale

By purchase of the software product described in this manual, you have obtained a license to duplicate the supplied disk files only as necessary for personal use on your Model 4 computer. None of the supplied files may be reproduced for resale.

If you intend to sell application programs developed using TRS-80 C, you must follow the procedure below to avoid violation of this license and of copyright laws.

1. Use the C compiler to translate the application program to object code.
2. Use the LINKLOAD utility to link the object code with the TRS-80 C runtime support and build a stand alone, executable command file (/CMD extension).
3. The executable command file may be copied and sold with no royalty payments required. However, all programs sold must document the fact that they contain TRS-80 C RUNTIME SUPPORT.

An Overview of the TRS-80 C Manual

There are six sections to this manual. It is suggested that you read through the Beginners Guide carefully. The six sections are:

(1) BEGINNERS GUIDE

1. Takes you through the steps of backing up the system.
2. Leads you through the steps of entering and executing a simple C program.
3. Trouble shooting guide.

(2) EDITOR MANUAL

Shows how to use the Blaise II text editor in detail.

(3) SYSTEM IMPLEMENTATION MANUAL

Gives specific information on the TRS-80 Model 4 implementation of C.

Included is more detailed information on:

1. Compiling and executing programs.
2. Memory usage.
3. Using the system dependent library functions.
(low level I/O routines, graphics, keyboard, and system call interfaces)
4. Using the library of dynamic string functions.
5. Using the random file functions.
6. Interfacing machine language programs to C.
7. Miscellaneous patches to modify the system.

(4) TUTORIAL

A step by step introduction to C aimed at people with some knowledge of a computer language.

(5) REFERENCE MANUAL

A detailed guide for the C language.

(6) ADVANCED DEVELOPMENT PACKAGE

Contains sections on the use and execution of the Codegen and Optimize programs. Explains when and why to use these utilities.

Disk Files

The TRS-80 C system includes the following files:

Disk 1 of 3

CC/CMD	C compiler
CMD/HLP	Editor help file
CERRORS/DAT	Error message file used by the compiler
EDIT/CMD	Text editor
HELP/HLP	Editor help file
KEY/HLP	Editor help file
RUNC/CMD	Fast load and run utility
SAMPLE/EDT	Sample binary setup file from Editor Manual
SETEDIT/CMD	Editor setup file utility
SETUP/EDT	Editor binary setup file
STDIO	Standard I/O header file
SYSTEM1/JCL	System file configuration
SYSTEM2/JCL	System file configuration

Disk 2 of 3

CODEGEN/CMD	Native code generator
CODEINIT/DAT	Data file for CODEGEN/CMD
CLIB/C	C source library
CLIB/OBJ	Object for C source library
LINKLOAD/CMD	Linking loader utility
OPTIMIZE/CMD	P-code Optimizer
PRINTF/C	C source (formatted output functions)
PRINTF/OBJ	Object (formatted output functions)
RANDOM/OBJ	Object (random file functions)
SCANF/C	C source (formatted input functions)
SCANF/OBJ	Object (formatted input functions)
STRINGS/OBJ	Object (dynamic string functions)
SYSTEM/OBJ	Object (low level system functions)
SYSTEM3/JCL	System file configuration
SYSTEM4/JCL	System file configuration
TRSLIB/OBJ	Object (TRS-80 specific functions)

Disk 3 of 3

CCB/CMD	Overlaid C compiler (for larger programs)
CC/OV1	Overlay 1 for overlaid C compiler
CC/OV2	Overlay 2 for overlaid C compiler
CC/OV3	Overlay 3 for overlaid C compiler
CC/OV4	Overlay 4 for overlaid C compiler
RUNC/OBJ	Contains TRSLIB, RANDOM, and STRINGS libraries
HEXTOBIN/CMD	Utility to convert hex files to binary
CSUPPORT/BIN	Contains all of the C libraries

Table of Contents

Chapter 1 Getting Started	1
1.1 Making Backups	1
1.2 File Configuration	2
1.2.1 Hard Disk Users	2
1.2.2 Floppy Disk Users	2
1.3 Overall System View	4
Chapter 2 Using the Editor	5
2.1 Editor Description	5
2.1.1 Setup Files	5
2.1.2 Using SAMPLE/EDT	6
2.1.3 Executing the Editor	6
2.2 Editor Commands	7
2.2.1 Cursor Movement	7
2.2.2 Insert and Delete	8
2.2.3 Other Commands	9
Chapter 3 Program Development	13
3.1 Editing	13
3.2 Compiling	14
3.3 Running	16
Chapter 4 Miscellaneous	17
4.1 File Names and Devices	17
4.2 Alternate Symbols	17
4.3 Standard Header File	18
4.4 Trouble Shooting	19
4.5 Common Error Messages	20
4.6 Common Programming Mistakes	21

Chapter 1

Getting Started

1.1 Making Backups

The first thing you should do before using the TRS-80 C system is to make backup copies of the three supplied master disks. Follow the steps below to create your backup disks.

- 1) Insert a TRSDOS 6 operating system disk into drive 0 and press the reset key.
- 2) When prompted with Date ? , type in the current date in the form mm/dd/yy and press the <enter> key. The screen will then display TRSDOS Ready.
- 3) Insert a new blank disk into drive 1 and type format :1 <enter>. Answer the prompts as follows:

Diskette name ? Disk1 <enter>
Master password ? password <enter>
Single or Double density <S,D> ? d <enter>
Number of cylinders ? 40 <enter>

The operating system will now format the disk and display the message "Formatting complete" when finished.
- 4) Now type: backup :0 :1 (x) <enter>
When prompted with Insert SOURCE disk <enter>, insert the TRS-80 C disk labeled "Disk 1 of 3" into drive 0 and press the <enter> key. The operating system will make a backup and then display the message: Insert SYSTEM disk <enter>. Insert the TRSDOS 6 operating system disk back into drive 0 and press the <enter> key.
- 6) Repeat steps 3 and 4 using the diskette name Disk2 instead of Disk1 in step 3 and "Disk 2 of 3" instead of "Disk 1 of 3" in step 4.

- 7) Repeat steps 3 and 4 using the diskette name Disk3 instead of Disk1 in step 3 and "Disk 3 of 3" instead of "Disk 1 of 3" in step 4.
- 8) Label the backup disks as Disk1, Disk2, and Disk3. Place your master TRS-80 C disks in a safe place and use the backup copies.

1.2 File Configuration

The C system files should now be arranged to provide a useful configuration for program development. How the files are arranged depends on the drive configuration of your Model 4.

1.2.1 Hard Disk Users

If you have a hard disk drive, then one useful configuration is to copy all the TRS-80 C files onto drive 0 of the hard disk. This may be accomplished by using the backup by class command after booting the hard disk system. Place the disk labeled Disk1 into one of the floppy drives. If the floppy drive number is 2, then the following command may be used to copy all the Disk1 files to drive 0 of the hard disk.

```
TRSDOS Ready  
backup :2 :0 (new) <ENTER>
```

Repeat the process using the disks labeled Disk2 and Disk3.

1.2.2 Floppy Disk Users

There are many ways of arranging the supplied C files to provide a suitable configuration for program development. How you arrange the files is dependent on the number of drives available.

If you have more than two floppy drives, a useful configuration would be to use drive 0 for the operating system, drive 1 for the Disk1, Disk2 or Disk3 disks, and the remaining drives for storing the programs being developed.

If you have only two drives, follow the configuration steps outlined on the next page. This is a sample 4 disk configuration that combines only the necessary operating system files with selected C files. With this configuration, drive 0 will be used as a system disk (containing the necessary operating system files and selected C files) and drive 1 will be used as a data disk (containing the programs being developed).

Configuration for a 2 drive system

- step 1) Make 4 backup copies of your original TRSDOS Version 6 operating system disk and label the backup copies as SYSTEM1 through SYSTEM4.
- step 2) Insert SYSTEM1 into drive 0 and Disk1 into drive 1.
 Type: DO =SYSTEM1 <ENTER>
 All unnecessary operating system files will be deleted from SYSTEM1 and the following C files will be copied from Disk1 to SYSTEM1: (CC/CMD, CERRORS/DAT, STDIO, RUNC/CMD, EDIT/CMD, SETUP/EDT, SAMPLE/EDT, CMD/HLP, HELP/HLP, KEY/HLP)
- step 3) Insert SYSTEM2 into drive 0.
 Type: DO =SYSTEM2 <ENTER>
 All unnecessary operating system files will be deleted from SYSTEM2 and the following C files will be copied from Disk1 to SYSTEM2: (SETEDIT/CMD, EDIT/CMD, HELP/HLP, CMD/HLP, KEY/HLP, SETUP/EDT, SAMPLE/EDT)
- step 4) Insert SYSTEM3 into drive 0 and Disk2 into drive 1.
 Type: DO =SYSTEM3 <ENTER>
 All unnecessary operating system files will be deleted from SYSTEM3 and the following C files will be copied from Disk2 to SYSTEM3: (LINKLOAD/CMD, CLIB/C, CLIB/OBJ, PRINTF/C, PRINTF/OBJ, SCANF/C, SCANF/OBJ, SYSTEM/OBJ, TRSLIB/OBJ, STRINGS/OBJ, RANDOM/OBJ)
- step 5) Insert SYSTEM4 into drive 0.
 Type: DO =SYSTEM4 <ENTER>
 All unnecessary operating system files will be deleted from SYSTEM4 and the following C files will be copied from Disk2 to SYSTEM4: (CODEGEN/CMD, OPTIMIZE/CMD, CODEINIT/DAT)

The disk labeled SYSTEM1 contains all the C files which are necessary to edit, compile, and execute programs. This is the only system disk needed for beginning programmers. The programs on SYSTEM2 through SYSTEM4 are for more advanced programming.

The disk labeled SYSTEM2 contains a utility for creating customized setup files for the editor. It also contains the editor and the associated help files and setup files.

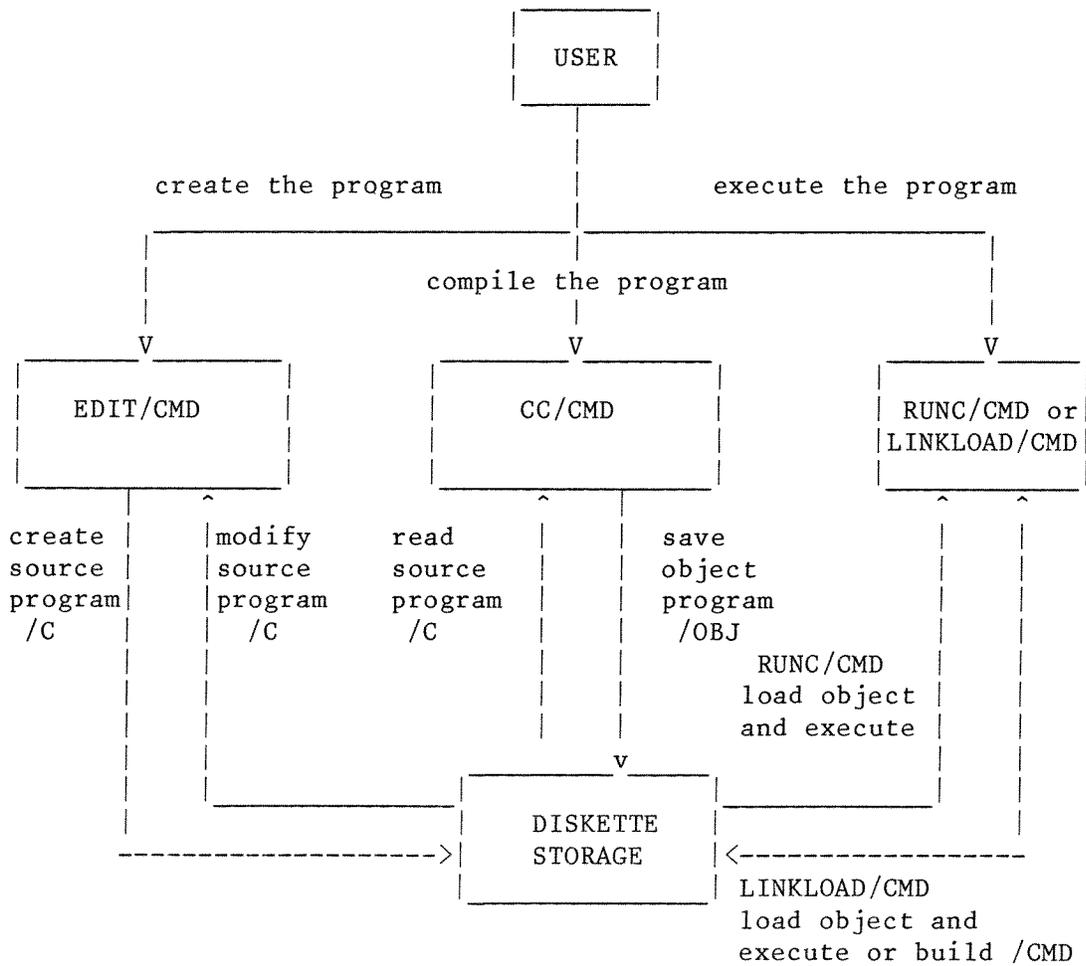
The disk labeled SYSTEM3 contains the linking loader and libraries. The linking loader must be used to link together separately compiled C functions. The object code files contain libraries of functions that a C program may call. The standard C functions (SYSTEM/OBJ, CLIB/OBJ, PRINTF/OBJ, and SCANF/OBJ) are built into RUNC/CMD but must be explicitly loaded from these

libraries when using LINKLOAD/CMD. The C source is provided for three of the libraries.

The disk labeled SYSTEM4 contains utilities for optimizing once you become familiar with the system. The files on this disk are explained further in the System Implementation Manual and the ADP Manual.

1.3 Overall System View

The following diagram illustrates the program development process using the TRS-80 C system.



Chapter 2

Using the Editor

The remainder of this manual describes the steps of editing, compiling, and executing a C program. For the following discussion, it is assumed that the files contained on the disk labeled SYSTEM1 (described in chapter 1) are available. If you have a two drive system, place the SYSTEM1 disk in drive 0. If you have more than two drives or if you have a hard disk, make sure that the files on SYSTEM1 are available on some drive. You should also have a formatted disk with plenty of free space for storing programs. If you have a two drive system, insert a formatted data disk into drive 1.

2.1 Editor Description

The editor has many commands which are internally mapped to standard ASCII control codes. These codes are generated from the keyboard by holding down the key labeled CTRL while pressing an alphabetic key. For example, the editor command to move the cursor one character to the right is mapped to CTRL D. (the Editor Manual contains a complete listing of internally mapped commands). An interesting feature of this editor is that the internal mapping of commands to keys may be changed. This means that you can design the keyboard layout to suit your own personal preferences. As an example, you may want to use the right arrow key (rather than CTRL D) to move the cursor one character to the right.

2.1.1 Setup Files

Each time the editor is executed, it reads a file named SETUP/EDT. The editor uses this file to determine how to operate. At a minimum, this file must contain information about the Model 4 terminal. For example, the editor must know how to position the cursor on the screen. Optionally, the file may contain information about how editor commands are mapped to keys. For example, it may tell the editor that the right arrow key should cause the cursor to move to the right.

The supplied file named SETUP/EDT contains only information about the Model 4 terminal. This setup file will cause the editor to only understand the internally defined mapping of commands to keys. In other words, the CTRL key is used to execute commands. The supplied file named SAMPLE/EDT is a sample editor setup file that contains the same terminal information but in addition defines a keyboard layout that utilizes the Model 4 arrow keys.

2.1.2 Using SAMPLE/EDT

The editor manual describes the operation of the editor based on the internal mapping of commands. At the end of the manual is a sample setup file that defines a keyboard layout that utilizes the Model 4 arrow keys. This is the supplied SAMPLE/EDT setup file. To illustrate how the editor's operation may be altered, this manual will describe the editor commands based on using the SAMPLE/EDT setup file. Using SAMPLE/EDT causes many of the internal editor command mappings to change. For example, the tab command (TB) is changed from CTRL I to CLEAR RIGHT ARROW.

Since the editor automatically reads the setup file named SETUP/EDT, you must rename the setup files in order to use SAMPLE/EDT. If the disk is write protected (tab covers write protect notch), the write protect tab must be removed before renaming the files. Type the following commands from the TRSDOS Ready prompt.

```
RENAME SETUP/EDT TO SETUP/SAV <ENTER>  
RENAME SAMPLE/EDT TO SETUP/EDT <ENTER>
```

2.1.3 Executing the Editor

Before executing the editor, make sure that there is plenty of disk space for storing files. It is a good practice to write protect the disks which are used as system disks (for example, SYSTEM1) to prevent data from being stored on them. On a two drive system, this will force all files to be stored on the data disk in drive 1.

The editor may be executed from TRSDOS Ready by typing a command of the following form.

```
EDIT filename <ENTER>
```

The filename is optional. If no file is specified, the editor will create a new file. The name of the new file will be specified at the end of the edit session. If a file name is specified, it should be the name of an ASCII formatted text file with record lengths of 80 characters or less. The file name may be any legal TRSDOS file name, including drive specifier. It is suggested that you specify a drive number with the file name. This will cause the editor to place the file on that drive when the editor is exited.

The editor reserves a section of memory which is used as a buffer for storing text. The symbol *EOB is displayed by the editor to indicate the "end of buffer". If no file is specified when the editor is executed, the buffer will start out empty and the *EOB symbol will appear at the top left corner of the screen. If a file is specified, the editor will load in the first 100 lines of the file and display a screen full of lines starting with the first line loaded.

If the buffer is empty, blank lines must be inserted into the buffer before text may be entered. Each time <CLEAR N> is typed (holding the CLEAR key down while pressing the N key), a blank line will be inserted into the text buffer. Once the buffer has lines in it, you may simply type in the text. Typing <LEFT ARROW> will cause the cursor to backspace if you need to correct a typing error. The <ENTER> key will cause the cursor to be positioned to the beginning of the next line. The editor will not allow the cursor to be positioned beyond the *EOB symbol. To enter more text after reaching the end of buffer, type <CLEAR N> to enter more blank lines into the buffer.

2.2 Editor Commands

The most often used editor commands are the ones which move the cursor around within the text buffer. Most of the cursor movement commands are mapped to the arrow keys. Other commands are mapped to the alphabetic keys and function keys.

2.2.1 Cursor Movement

There are four basic cursor movement commands. (right, left, up, and down). Each of these commands moves the cursor in the specified direction. These commands have been mapped to the arrow keys. They are executed by simply pressing the appropriate arrow key.

Key	Command Name	Function
<RIGHT ARROW>	RT (right)	move cursor right 1 character
<LEFT ARROW>	LF (left)	move cursor left 1 character
<UP ARROW>	UP (up)	move cursor up 1 line
<DOWN ARROW>	DN (down)	move cursor down 1 line

The basic cursor movement commands provide the ability to position the cursor any place on the screen. However, moving only a single character or line at a time can be a little slow. Other commands are mapped to allow you to move the cursor more efficiently.

There are two commands that move the cursor left or right by one tab stop. The tab command moves the cursor to the right to the next tab stop. The back tab command moves the cursor to the left to the next tab stop.

Since the text buffer holds more than a screen full of text, you also need a way to scroll back and forth in the buffer. The roll up command moves the

cursor one screen towards the beginning of the buffer while the roll down command moves the cursor one screen towards the end of the buffer.

These commands have also been mapped to the arrow keys. They are executed by holding down the CLEAR key while pressing the appropriate arrow key.

Key	Command Name	Function
<CLEAR RIGHT ARROW>	TB (tab)	move cursor right to the next tab stop
<CLEAR LEFT ARROW>	BT (back tab)	move cursor left to the next tab stop
<CLEAR UP ARROW>	RU (roll up)	move one screen toward the top of the buffer
<CLEAR DOWN ARROW>	DN (roll down)	move one screen toward the bottom of the buffer

Other commands provide the ability to move the cursor greater distances even more efficiently. The beginning of line command positions the cursor at the beginning of the line. The end of line command positions the cursor at the end of the line. The top of buffer command displays the first line in the buffer at the top line of the screen. The bottom of buffer command positions the cursor at the *EOB mark at the end of the buffer.

These commands are also mapped to the arrow keys. They are executed by pressing and releasing the BREAK key and then pressing the appropriate arrow key.

Key	Command Name	Function
<BREAK RIGHT ARROW>	EL (end line)	move cursor to the end of the line
<BREAK LEFT ARROW>	BL (beginning line)	move cursor to the beginning of the line
<BREAK UP ARROW>	TP (top of buffer)	display the first line in the buffer at top of screen
<BREAK DOWN ARROW>	BB (bottom of buffer)	move the cursor to the *EOB mark at the end of buffer

2.2.2 Insert and Delete

Seven commands are mapped to alphabetic keys. You have already used one, the insert line command. There are three commands that delete either a character, word, or line.

The undelete line command may be used to restore a line that was accidentally removed by the delete line command. There is also a duplicate line command that may be used to make a duplicate copy of the line above the cursor.

The insert character command may be used to insert characters in a line. When this command is executed, subsequent characters that you type will be inserted at the current cursor position. The editor will continue to insert characters until a non-printable character (such as the <ENTER> key) is typed.

These commands are mapped to alphabetic keys. They are executed by holding down the CLEAR key while pressing the appropriate alphabetic key.

Key	Command Name	Function
<CLEAR N>	IL (insert line)	insert a blank line at the cursor line
<CLEAR C>	DC (delete character)	delete character under the cursor
<CLEAR W>	DW (delete word)	delete word under the cursor
<CLEAR L>	DL (delete line)	delete line under the cursor
<CLEAR U>	UL (undelete line)	restore the last deleted line
<CLEAR D>	DU (duplicate line)	duplicate the line above the cursor
<CLEAR I>	IC (insert character)	insert characters until a non-printable is typed

2.2.3 Other Commands

Six other frequently used commands are mapped to the Model 4 special function keys (F1 through F3).

The forward word command moves the cursor to the first character of the word to the right. The backward word command moves the cursor to the first character of the word to the left. Both the forward word and backward word commands will move the cursor across line boundaries.

The split line command creates two lines out of one. This command causes all characters to the right of the cursor to be moved to a new line below. The merge line command is used to merge two lines. As many characters as will fit on a line are moved from the line below the cursor to the end of the line containing the cursor.

The insert mode command is similar to the insert character command. However, it does not terminate when a non-printable character is typed. The editor continues to insert characters until the insert mode command is executed again. This command toggles the editor in and out of insert mode. While in insert mode, the editor inserts a blank line when the <ENTER> key is typed. If the <ENTER> key is typed in the middle of a line, the characters to the right of the cursor are moved to the next line.

The last command that you must know is the command that places the editor in command mode. Command mode allows all editor commands to be executed. This is important since not all commands are mapped to a key. When this command is executed, the editor displays angle brackets <> at the bottom left corner of the screen. Then any editor command may be executed by typing its two character command name followed by the <ENTER> key. For example, UP <ENTER> would execute the cursor up command and then exit command mode.

These commands are executed by pressing the appropriate function key. Three of them require that you hold down the shift key while pressing the function key.

Key	Command Name	Function
<F1>	CM (command mode)	enter command mode
<F2>	BW (backward word)	move the cursor left one word
<F3>	FW (forward word)	move the cursor right one word
<SHIFT F1>	IM (insert mode)	enter permanent insert character mode
<SHIFT F2>	SP (split line)	split the line at the cursor
<SHIFT F3>	MG (merge line)	merge the line below with the cursor line

The commands described so far should be quite adequate for handling most of your editing needs. The editor has many other commands which are described in the Editor Manual. Once you become familiar with these commands, you will want to read the Editor Manual for information on other available commands.

For now, the only other commands you must know are the commands to terminate an edit session. To execute these commands, you must enter command mode. As described earlier, pressing the <F1> function key puts the editor in command mode.

Two commands may be used to terminate an edit session. The first command is EX (exit). This command should be used if you wish to save the text to a file. The other command is QT (quit). This command should be used if you wish to terminate the edit session without saving the text.

The EX command requires two parameters, the name of the file to which the text will be written, and whether or not you wish the editor to create a backup file. The editor will prompt you to enter both of these parameters when EX <ENTER> is typed. The first prompt is for the file name. If creating a new file, now is the time that the file name must be specified. Simply type in a valid file name. If editing a pre-existing file, you may simply type <ENTER> to the file name prompt. The text will be written to the file specified when the editor was executed. The second prompt is whether or not

to create a backup file. You may answer this prompt by typing either Y for yes or N for no, followed by the <ENTER> key. Simply typing the <ENTER> key for this prompt is equivalent to typing Y <ENTER>. A backup file is created only if the file being edited already exists. The file specified in the EXIT command is renamed with the extension /BAK before the new file is written out. The backup file may be used to restore a file if the file is for some reason damaged. The backup file will reflect one edit session prior to the current one.

The QT command is used to terminate the edit session without saving anything. Simply type QT <enter>. You will then be prompted to make sure that this is what you really want to do. If you answer Y <enter>, the editor terminates. Otherwise, the edit session is continued.

Chapter 3

Program Development

3.1 Editing

Now that you know how to use the editor, a simple C program may be created. Drive 1 will be used to store the program so make sure that there is plenty of free disk space on drive 1 before beginning. First type EDIT <enter> and the editor is executed. Since no file was specified, *EOB will appear at the top left corner of the screen. Type CLEAR N four times to enter four blank lines into the buffer.

Note: The { symbol is generated by "shift clear <".
The } symbol is generated by "shift clear >".

Type in the following text.

```
main()
{
    printf(" This is my first program.")
}
*EOB
```

Once the text has been entered as shown above, press <F1> to enter command mode. Execute the exit command and answer the two prompts as shown below.

```
<> EX <enter>
<EXIT>FILE: TEST/C:1 <enter>
<EXIT>BACKUP? <enter>
```

The program will be saved to the file TEST/C on drive 1 and the editor will exit to the operating system. It is important to name your C source files with the extension /C because the compiler uses this as the default extension.

3.2 Compiling

The compiler must now be used to translate the C program to object format. Once in object format, the program may be executed. Type the following to compile the program created in the previous chapter.

```
CC TEST:1 <enter>
```

The C compiler will execute and begin reading the file TEST/C on drive 1. As each line is read, it is translated to object code. The compiler will write the object code to the file TEST/OBJ on drive 1. The compiler also sends a listing to the screen as it compiles. The listing will show if there are any errors in the program being compiled. The below listing was generated by compiling the sample test program.

```
TRS80 C VERSION: 02.01.00                xx:xx:xx xx/xx/xx PAGE 1
-----
 1|main()
 2|{
 3|  printf(" This is my first program.")
 4|}
***** ^14, 2
ERRORS DETECTED

      2 ERRORS DETECTED

      2 IDENTIFIER OR OTHER LVALUE EXPECTED
      14 ';' EXPECTED

STACK USED =  xxxx OF xxxx    HEAP USED =  xxxx OF xxxx
```

As you can see, the compiler detected some errors in the program. The compiler always writes an error message line following the line where the error was detected. The error message line begins with 5 asterisks to clearly indicate that an error was detected. It also contains a pointer to the line above at the approximate location of the error. Following the pointer is an error code telling the type of error detected. At the end of the listing, all generated error codes are listed with a brief explanation of the error.

All C programs, including the compiler, use a section of memory which is divided into two parts. One part is the stack and the other part is the heap. The stack is used to store most variables. The heap is used to store dynamic variables and file descriptors. When the compile is finished, the amount of stack and heap used out of the total amount available is displayed on the screen.

Because of the context in a C program, a single error in the program can generate multiple error messages. Usually, the first error code will describe the real cause of the error. In this example, the first error detected is on line 4, error code 14. Error code 14 says that a semicolon was expected. This error was caused by the failure to terminate the statement on line 3 with a semicolon. The other error, error code 2, is a side effect of the first error. The compiler automatically creates a file named

C/ERR

when errors are detected in a program. Only the lines containing errors, along with the error message line, are written to this file.

The program should now be corrected before it is executed. The following will cause the editor to execute and display file TEST/C on the screen.

EDIT TEST/C:1 <enter>

Move the cursor to the third line of the program and add a semicolon just after the right parenthesis. The third line should then look as follows.

```
printf(" This is my first program.");
```

Press <F1> to enter command mode and type EX to exit the editor. The two prompts may be answered by simply pressing the <enter> key.

Now the program may be compiled once again by typing the following.

CC TEST:1 <enter>

The following listing will be sent to the screen as the program is compiled.

```
TRS80 C  VERSION: 02.01.00          xx:xx:xx xx/xx/xx  PAGE 1
-----
 1|main()
 2|{
 3| printf(" This is my first program.");
 4|}
TEST
NO ERRORS DETECTED
NO ERRORS DETECTED

STACK USED = xxxx OF xxxx  HEAP USED = xxxx OF xxxx
```

This time no errors were detected. The program is a legal C program. Now that the program has been compiled with no errors, it may be executed.

3.3 Running

Once the program has been compiled without errors, it can be executed with the RUNC command. From the TRSDOS Ready prompt, type the following to execute the program in file TEST/OBJ on drive 1.

```
RUNC TEST:1 <enter>
```

The program will print the following message on the display screen (also referred to as the "crt").

```
This is my first program.
```

When a program terminates, (ie. finishes execution normally), the address of the last instruction executed is displayed on the screen. Following this is the amount of stack and heap used by the program. The stack and heap are explained in the System Implementation Manual. The miscellaneous patch section of the System Implementation Manual also explains how to prevent this information from being displayed.

C programs perform input and output using file pointers. A file pointer may point to a device such as the "crt" or to a disk file such as "test/dat". When a C program is executed, two standard input and output file pointers are automatically defined. These are called "stdin" and "stdout". By default, "stdin" points to the keyboard while "stdout" points to the crt. Some of the input library functions are defined to receive input from "stdin" while some of the output library functions are defined to send output to "stdout". The "printf" function sends its output to "stdout". Therefore, our sample program sent the message to the crt.

It is possible to change the "stdin" and "stdout" file pointers so that they point to a disk file or some other device when a program is executed. This is called IO redirection. The RUNC command allows you to redirect the file pointers "stdin" and "stdout". The < symbol is used to redirect the input file pointer "stdin" and the > symbol is used to redirect the output file pointer "stdout". Our sample program may be executed again with the message being sent to a disk file rather than to the crt. For example, the following command will execute the sample program with the output going to the file TEST/DAT on drive 1.

```
RUNC TEST:1 >TEST/DAT:1 <enter>
```

Chapter 4

Miscellaneous

4.1 File Names and Devices

The C compiler always uses the extension /C if the file name is specified when the compiler is executed. The compiler may also be executed by simply typing CC without a file name. If executed in this manner, the compiler will prompt for the C SOURCE file name, the file to use for the LISTING, and the file to use for the OBJECT. Either a file name or a device may be specified. If a file name is specified, the complete file name, including extension, must be used (ie. the compiler does not use default extensions). The RUNC utility uses the default extension /OBJ if no extension is specified in the file name. You may also specify an extension if the object code is in a file named with an extension other than /OBJ. For example, RUNC TEST/COD might be used.

The file names that you use to direct C input and output are the same format as normal TRSDOS file names. The disk drive specification is optional. Devices may also be specified instead of a file name. For example, the name of the line printer is ":L". The name of the terminal which is the keyboard for input and the crt for output, is ":C". There is also a dummy device. If a file is associated with ":D", then no actual output occurs. This is useful if you wish to discard certain outputs. For example, the listing may be discarded during a compile or you might discard some of the output generated by a program when it is executed. The sample program may be compiled with the listing turned off and executed with the output being sent to the line printer.

<u>Compile</u>	<u>Execute</u>
CC	RUNC TEST:1 >:L
SOURCE = TEST/C:1	
LISTING = :D	
OBJECT = TEST/OBJ:1	

4.2 Alternate Symbols

The C compiler recognizes alternate representations of certain symbols

because not all terminals have the ability to generate them. The alternates may be used in place of the normal C symbols if desired. However, it is best to use the normal symbols to maintain compatibility with standard C notation.

symbol -----	Generated on Model 4 by -----	alternate -----
{	clear shift <	(#
}	clear shift >	#)
	clear shift /	/!
	-----	/!!
^	clear ;	@
[clear <	(@
]	clear >	@)
\	clear /	//
~	-----	!!
_	clear enter	-----

4.3 Standard Header File

The standard header file, `STDIO`, contains the definition of types, constants, and variables which are used by the C function libraries. Most of the definitions are related to file input and output.

```
/****** Standard I/O Header *****/
```

```
#define MAXFILES 20
#define EOF -1
#define NULL 0
#define stdin _iob[0]
#define stdout _iob[1]
#define stderr _iob[2]
#define getchar() getc(stdin)
#define putchar(c) putc(c, stdout)
```

```
typedef struct {
    char file[32]; /* file descriptor */
    int fd; /* file descriptor number */
} FILE;
```

```
extern FILE *_iob[MAXFILES];
```

```
typedef struct { /* dynamic string structure */
    int length;
    char string[80];
} STRING;
```

```
/******
```

Generally this file should be included in the compilation of a C program. Any C program that uses an identifier that is defined in STDIO must include this file. Another file may be included in the compilation of a C program by using the #include command. For example, our sample program can include the standard header file in the compilation as follows.

```
#include "stdio"
main()
{
    printf(" This is my first program.");
}
```

4.4 Trouble Shooting

(Miscellaneous Errors)

1. Problem - While editing a file, the latter part of a file is found to be missing.

Answer - Need to use the APPEND command to page the latter

part of the file into the text buffer. See the Editor Manual.

2. Problem - Upon exiting the editor, a PHYSICAL IO error message is displayed.

Answer - The disk is full. Make sure that there is plenty of free disk space when editing files.

3. Problem - During a compile, the C compiler abnormally terminates with a FATAL ERROR - OUT OF HEAP, or OUT OF STACK

Answer - The compiler does not have enough memory for either the stack or the heap. Specify the size of the stack during the compile (eg. CC <5K> TEST) or split the program into separately compiled files.

4. Problem - When executing your compiled program with the RUNC command, or a command (/CMD) file built with the LINKLOAD utility, it abnormally terminates with the FATAL ERROR - OUT OF HEAP, or OUT OF STACK

Answer - Specify the amount of stack when using the RUNC command or the build command of the linking loader. (eg. RUNC TEST 10K : See the System Implementation Manual for further details).

5. Problem - After executing the compiler using the long form where the OBJECT and LISTING files are specified, the original source file suddenly contains object code.

Answer - The /C extension was used when specifying the object file.

4.5 Common Error Messages

(By the Compiler)

- 2 IDENTIFIER OR OTHER LVALUE EXPECTED - This error code often follows a prior error code because the compiler begins looking for the next statement.
- 13 '}' EXPECTED - There must be a } for every { in a C program.
- 14 ';' EXPECTED - The preceding declaration or statement is not terminated by a semicolon (;).

- 104 UNDECLARED IDENTIFIER - All variables must be declared in a C program.
- 129 TYPE CONFLICT OF OPERANDS IN AN EXPRESSION - Using incompatible operand types in an expression.

(Runtime Error Messages)

The error codes discussed above are generated by the compiler due to an error in the C source program. There are times when the compiler may generate a fatal error message that is not due to an error in the source program. These are called runtime errors because they are detected by the runtime that is included with all C programs, including the compiler. The following are examples of runtime errors.

RUNTIME ERROR 01 OUT OF STACK
(Caused by trying to compile or run too large of a program)
RUNTIME ERROR 02 OUT OF HEAP
(Caused by trying too compile or run too large of a program)
RUNTIME ERROR 09
(file not found or disk error)

Note: Explanation of COMPILER and RUNTIME error codes may be found in the appendix of the Reference Manual.

4.6 Common Programming Mistakes

1. The assignment operator = is used rather than the equality operator == in a conditional expression. For example, in the statement below, a = b will never be executed because the expression (i = 0) always evaluates to 0 (false).

used: if (i = 0) a = b;
intended: if (i == 0) a = b;

2. A variable containing a character is compared with an integer constant rather than a character constant.

used: c = getchar();
if (c >= 0 && c <= 9) printf(" digit");
intended: c = getchar();
if (c >= '0' && c <= '9') printf(" digit");

3. The library functions such as getchar return characters as integers so that EOF may be returned. If a variable

declared of type char is used to store a returned character, comparison with EOF will always be false.

```
used: char c;
      c = getchar();
      if (c == EOF) exit;
      printf("EOF detected");
intended: int c;
          c = getchar();
          if (c == EOF) exit;
          printf("EOF detected");
```

4. The indexes used to access the elements of an array are off by 1. The declaration "char a[20];" creates an array of 20 characters with an index range of 0..19 rather than 1..20.

```
used: for (i=1; i <= 20; i++) a[i] = 'z';
intended: for (i=0; i <= 19; i++) a[i] = 'z';
```

5. The break statement is omitted at the end of one of the case labels of a switch statement. Without a break statement, execution falls through to the next case label.

```
used: switch (c) {
      case 'a': printf("value of c is 'a'");
      case 'b': printf("value of c is 'b'");
      }
intended: switch (c) {
          case 'a': printf("value of c is 'a'");
                  break;
          case 'b': printf("value of c is 'b'");
      }
```

Table of Contents

Chapter 1 Blaise II Overview	3
1.1 Setup Files	3
1.2 The SETEDIT Program	4
1.3 Text Buffer Management	5
1.4 The Work File	6
1.5 Compose Mode	6
1.6 Command Mode	7
Chapter 2 Getting Started	9
2.1 Editor File Configuration	9
2.2 Terminal Configuration	9
2.3 Executing the Editor	12
2.4 Basic Editor Commands	13
2.5 Editor Help Files	17
2.6 Swapping Disks	17
2.7 Exiting the Editor	18
2.8 Sample Edit Session	19
Chapter 3 Editor Commands	21
3.1 Command Parameters	21
3.2 Cursor Positioning Commands	23
3.2.1 New Line [NL]	23
3.2.2 Right [RT]	23
3.2.3 Left [LF]	23
3.2.4 Up [UP]	23
3.2.5 Down [DN]	23
3.2.6 Tab [TB]	23
3.2.7 Back Tab [BT]	24
3.2.8 Forward Word [FW]	24
3.2.9 Backward Word [BW]	24
3.2.10 End of Line [EL]	24
3.2.11 Beginning of Line [BL]	24
3.2.12 Home [HM]	24
3.2.13 Roll Up [RU]	24
3.2.14 Roll Down [RD]	24
3.2.15 Top of Buffer [TP]	25
3.2.16 Bottom of Buffer [BB]	25
3.2.17 Go to Mark [GM]	25
3.2.18 Swap [SW]	25
3.2.19 Set Row [SR or ROW]	25
3.2.20 Set Column [SC or COL]	25
3.2.21 Position [PO or POSITION]	25
3.2.22 Minus [MI or -]	26
3.2.23 Plus [PL or +]	26

3.2.24 Show Line(SL or SHOWLINE)	26
3.2.25 Horizontal Scroll [HS or HSCROLL]	26
3.3 Inserting Text	26
3.3.1 Insert Mode [IM]	27
3.3.2 Insert Character [IC]	27
3.3.3 Insert Line [IL]	27
3.3.4 Undelete Line [UL]	27
3.3.5 Quote [QU]	27
3.3.6 Quote String [QS or QUOTE]	28
3.4 Deleting Text	28
3.4.1 Delete Character [DC]	28
3.4.2 Rub Out [RB]	28
3.4.3 Delete Word [DW]	28
3.4.4 Delete Line [DL]	28
3.4.5 Delete to End [DE]	28
3.5 String Search and Replace	29
3.5.1 Find String [FS or FIND]	29
3.5.2 Replace String [RS or REPLACE]	29
3.5.3 Find Next [FN]	29
3.5.4 Replace Next [RN]	30
3.5.5 Replace Global [RG or REPGLOB]	30
3.6 Block Commands	30
3.6.1 Mark [MK]	30
3.6.2 Copy Block [CB]	30
3.6.3 Insert Block [IB]	31
3.6.4 Delete Block [DB]	31
3.6.5 Lower Case [LR]	31
3.6.6 Upper Case [UR]	31
3.6.7 Print [PR]	31
3.6.8 Fill [FI or FILL]	31
3.6.9 Justify [JF or JUSTIFY]	31
3.6.10 Extract [XT or EXTRACT]	32
3.7 File Commands	32
3.7.1 Help [HP or HELP]	32
3.7.2 Directory [DI or DIR]	32
3.7.3 Show File [SF or SHOWFILE]	32
3.7.4 Insert File [IF or INSFILE]	33
3.7.5 Delete File [DF or DELFILE]	33
3.7.6 Save [SV or SAVE]	33
3.7.7 Append [AP or APPEND]	33
3.7.8 Write [WR or WRITE]	34
3.8 Setting and Clearing Tab Stops	34
3.8.1 Delete Tabs [DT]	34

3.8.2 Set Tab [ST]	34
3.8.3 Clear Tab [CT]	34
3.8.4 Tab Stops [TS or TABS]	34
3.9 Miscellaneous	35
3.9.1 Command Mode [CM]	35
3.9.2 Duplicate [DU]	35
3.9.3 Merge [MG]	35
3.9.4 Split [SP]	35
3.9.5 Center Line [CL]	35
3.9.6 Auto Indent [AI]	35
3.9.7 Line Numbers [LN]	36
3.9.8 Memory [MM]	36
3.9.9 Refresh [RF]	36
3.9.10 Tabify [TF]	36
3.9.11 Swap Disk [SD]	36
3.9.12 Roll [RL or ROLL]	36
3.10 The EDIT Command [ED]	37
3.11 Terminating an Edit Session	37
3.11.1 Exit [EX or EXIT]	37
3.11.2 Exit/ [E/ or EXIT/]	38
3.11.3 QUIT [QT or QUIT]	38
3.11.4 QUIT/ [Q/ or QUIT/]	38
Chapter 4 Changing Editor Characteristics	39
4.1 Translating Keys to Commands	39
4.1.1 Translate [TR or TRANS]	40
4.2 Defining Macro Commands	40
4.2.1 Define Macro [DM or DEFINE]	40
4.2.2 Undefine Macro [UM or UNDEFINE]	42

Chapter 5 Editor Setup Files	43
5.1 Normal Commands	44
5.1.1 TABS	44
5.1.2 ROLL	44
5.1.3 AUTOINDENT	44
5.1.4 TRANS	44
5.1.5 DEFINE	44
5.2 Special Commands	45
5.2.1 INIT	45
5.2.2 EXIT	45
5.2.3 START	45
5.2.4 CMD	46
5.2.5 HEIGHT	46
5.2.6 WIDTH	46
5.2.7 TERMINAL	46
5.2.8 CURSOR	47
5.3 Sample Setup Files	48
Appendix A Custom Setup	55
A.1 Sample Terminal Setup	55
A.2 Special Cursor Addressing	56
Appendix B Standard 7-bit ASCII Character Set	59

Introduction

A text editor is simply a program that is used to enter text and save the text to a file. Usually, text editors are classified into one of two categories, line or screen editors.

Line editors operate on text a line at a time. Typically, you must view the text being edited by listing the lines that fall between two specified line numbers. Usually, the text must be modified by typing commands which operate on a specified line number. To change a character on a line, you often must use a command rather than being able to position the cursor to the bad character and correcting it.

Screen editors operate on a screen full of text at a time. A screen editor gives you much more context when editing a file. You are able to move the cursor around on the screen, changing the text by simply typing over the incorrect text. Rather than thinking in terms of line numbers, a screen editor allows you to scroll through the text a page at a time. A screen editor makes editing much easier by providing more powerful commands and an environment which allows you to see what you're changing as you change it.

The Blaise II text editor is a screen editor. It provides a very good tool for entering your programs or other textual documents. Some of the features found in word processors have been included in the Blaise II editor. Although it was designed for program entry, you may find that it serves many of your word processing needs as well.

Chapter 1

Blaise II Overview

This chapter provides a brief description of the major features of the Blaise II text editor. It should provide you with a basic understanding of how the editor operates. The actual use of the editor is more completely explained in the following chapters.

Blaise II is a very powerful text editor with many commands and features. It is designed so that you can change the characteristics of the editor to conform to personal preferences. For example, you can change how commands are mapped to keys. As supplied, the editor is internally configured to map the most frequently used commands to the keyboard. These commands are executed by typing control characters. It is suggested that the inexperienced user learn how to use the editor with this standard configuration. The more experienced user may want to alter the editor characteristics so that it operates similar to some other familiar editor.

1.1 Setup Files

Before using the editor, it must be configured for the type of terminal used by your computer. The editor uses what is called a setup file to perform this configuration. A setup file is simply a file containing commands that the editor understands. Each time the editor is executed, it reads the setup file and configures itself based on these commands. The commands in a setup file tell the editor about the terminal's smart features. For computers that use a known type of terminal, for example the Model 4 computer sold by Radio Shack, the setup file is supplied. It is named SETUP with an EDT extension. This is the default file that the editor loads if a setup file is not specified when the editor is executed. The supplied setup file contains only information about the commands that the terminal understands. If the SETUP file is not supplied, you must create it using the SETEDIT utility.

The setup file may also contain commands that cause the editor to default to some desired state. You may do things such as define how keys are mapped to editor commands, or define new commands which are composed from the set of built-in commands. It is often desirable to at least map the cursor movement commands to the arrow keys on your keyboard. If your keyboard has function keys, you may want to utilize them as well. There are no standard characters generated by arrow keys or function keys. In most cases, the character generated on one terminal will not be the same as that generated on a terminal of a different type. You will have to look in the documentation for your

particular terminal to see what characters are generated by these keys in order to map them to editor commands.

There may be a sample setup file included on your master disk. The sample setup file maps the terminal's arrow keys to cursor movement commands. Other commands may be mapped to provide better utilization of a specific keyboard.

1.2 The SETEDIT Program

The SETEDIT program is a utility used to create setup files for use with the editor. This utility provides a menu of commands which allow various setup file-related operations to be performed. If a setup file is not supplied with your master disk, this utility must be used to create the setup file before the editor can be executed. The SETEDIT utility displays the following menu.

```
T = define terminal characteristics
R = read a text format setup file
W = write a binary setup file
I = input a binary setup file
O = output a text setup file
H = display help information
E = exit
```

At a minimum, the editor requires a setup file that contains terminal information. This information tells the editor about the smart features of the terminal. The editor uses this information to display text in the most efficient way possible. The T command provides a list of commonly used terminals. If your system uses one of the listed terminals, then terminal definition is accomplished simply by selecting the proper terminal.

The editor requires binary formatted setup files. Once the proper terminal has been selected, a binary setup file may be created using the W command. The editor uses the file named SETUP (with an EDT extension) as the default setup file. When the editor is executed, it loads this file if no setup file is specified. Once the terminal information is written to this setup file, the editor can be used.

If you wish to view or modify the terminal information created by the SETEDIT utility, you may use the O command to output the information in text (readable) form. It is important to note that the editor can edit text formatted files only. However, the setup files used to configure the editor must be in binary format.

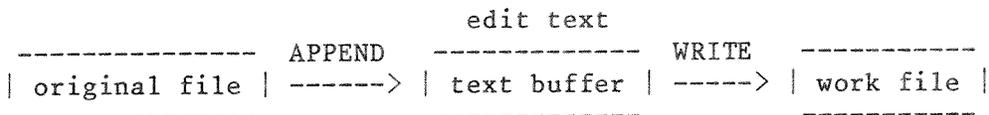
The SETEDIT utility has the ability to read either text or binary formatted setup files with the R and I commands respectively. A binary formatted setup file must have been previously created by SETEDIT. A text formatted file may be created using the editor. The SETEDIT utility may be used to combine

multiple setup files and terminal information into one single setup file. For example, after creating the binary setup file containing the terminal description only, the editor may be used to create a text formatted setup file containing other commands. Then the SETEDIT utility can be used to read both the files and write the combined information out to another file.

1.3 Text Buffer Management

The editor maintains a fixed size buffer for storing text. The buffer will hold approximately 15000 characters on a 64K system. All editor commands except for specific file commands operate only on the text in this buffer. When editing very large files, the file must be edited a section at a time. Starting at the beginning of the file, a section is loaded into the text buffer. Before loading another section of the file into the buffer, buffer space must be made available by writing the text out to a work file. Then the next section may be loaded into the buffer. This process may be repeated until the whole file has been loaded and edited.

When editing an existing file, the editor loads the first 100 lines only. This leaves ample buffer space for adding more lines and performing the various editing functions. If the file is longer than 100 lines, the APPEND command may be used to load more text from the file into the buffer. With this command, you specify how many lines to copy from the file to the buffer. The copying begins one line past the last line previously loaded from the file. The text being copied from the file is appended to the end of the text in the buffer. If the file is very large, it is possible for the buffer to become full. If this happens, a MEMORY EXHAUSTED message is displayed. The WRITE command must then be used to write some of the text in the buffer back out to a work file. With this command, you specify how many lines to copy from the buffer to the work file. The copying begins with the first line in the buffer and continues until either the buffer is empty or the specified number of lines have been written. Once lines have been written from the buffer to the work file, they may not be edited again during the current edit session. However, the EXIT/ command can be used to terminate an edit session if you wish to edit the lines that have already been written to the work file. The EXIT/ command leaves the editor loaded. Then the APPEND command can be used to load lines starting at the beginning of the file. The following diagram illustrates the editing process.



If the editor is exited before the entire file has been loaded into the buffer, the editor will copy the remaining lines in the original file to the work file. The work file is then renamed as the original and the original is either deleted or renamed as a backup file.

1.4 The Work File

The editor creates temporary files during an edit session. These temporary files are called work files. The main work file is a copy of the file being edited. Its purpose is to prevent a system crash from damaging the original file. If such a crash occurs during an edit session, the original file is left unchanged. The main work file is named T0n1 where n is either 1 or 2 depending on the level of the edit. The level is 1 when editing a single file. If the EDIT command is used during an edit session, a second work file is created, level 2. The extension TMP is appended to the work file names.

After successfully exiting, if the work file is on the same disk as the original file, then the work file is renamed as the original and the original file is either deleted or renamed as a backup file. If the work file is on a separate disk from the original, then it is copied over the original after which the work file is deleted.

The work file is placed on the same drive as the original file if a drive is specified with the file name when the editor is executed. If you are creating a new file and the WRITE command is used, the work file is placed on the default drive. The default drive is determined by the operating system. When the work file is on the same drive as the original, there must be enough free disk space to accommodate two copies of the original file.

The editor block movement commands also cause the editor to create a work file. This work file is used as temporary storage for the block of data being moved. The name for this work file is T0n3 where n is either 1 or 2 as above. The editor does not delete this work file.

1.5 Compose Mode

The editor has two modes of operation, compose mode and command mode. When the editor is executed, it starts out in compose mode. In this mode, you may type in text much the same way as you would with a typewriter. The text that you type is stored in the text buffer. While in this mode, you have access to many editor commands. For example, there are commands to move the cursor around on the screen. These commands are mapped to specific control characters which are generated from the keyboard. A control character is generated by holding down the key labeled CTRL while pressing an alphabetic key. For example, CTRL X will cause the cursor to move down one line, CTRL G deletes the character under the cursor, etc. Therefore, compose mode allows you to enter text and to move around in the text performing various operations using control characters.

1.6 Command Mode

One control character is mapped to a command that causes the editor to switch from compose mode to command mode. When CTRL Z is typed, the editor displays angle brackets <> at the bottom left corner of the screen and goes into command mode.

Command mode provides the ability to execute any command in the editor. Since not all editor commands are mapped to control keys, it is necessary to go into command mode to execute the unmapped commands. All commands in the editor, whether mapped to keys or not, have a two character mnemonic. While in command mode, the commands are executed by typing the two character mnemonic followed by the <enter> key.

Command mode provides a convenient alternate method of executing editor commands. The editor has so many different commands that it is impossible to map all commands to keys in a manner that is logical and easily remembered. A mnemonic is often easier to remember than a control character. The two character mnemonic reflects the actual function of a particular command while a control character sequence may not.

Command mode is entered from compose mode by typing CTRL Z. Angle brackets appear at the bottom left corner of the screen and the cursor is placed to the right of the brackets. Any command may then be executed by typing its two character mnemonic followed by the <enter> key. If there are typing errors, CTRL H may be used to backspace and make corrections. Once the command has finished execution, the editor returns to compose mode. The screen will reflect any changes caused by the execution of the command. To return to command mode, a CTRL Z must be typed once again.

A convenient way of operating the editor is to execute often used commands from compose mode and seldom used commands from command mode. For example, the cursor positioning commands (cursor right, cursor left, cursor up, cursor down) are used constantly. It would be inconvenient to execute these commands from command mode. On the other hand, the directory command (DI) is seldom used. Rather than try to remember what control sequence it is mapped to, it may be easier to remember the mnemonic DI and execute this command from command mode.

Chapter 2

Getting Started

The text editor is the command file named EDIT. There are also three help files named with the extension HLP. The help files contain information about the editor commands and may be viewed during an edit session. They are not necessary for the operation of the editor. They simply provide helpful information if needed. The command file named SETEDIT is used for creating editor setup files. The editor cannot be executed without a setup file being present. There may be an editor setup file named SETUP and possibly another setup file named SAMPLE. Both have EDT extensions. These files are supplied for computers with known terminal types. The SETUP file contains only terminal information. The SAMPLE setup file contains commands to remap some of the editor commands in addition to the terminal information.

Before beginning, be sure to make a backup copy of the supplied master disk. Place the master disk in a safe place and use the backup copy.

2.1 Editor File Configuration

The EDIT command file may be placed in any drive. On some operating systems (eg. CP/M), the default editor setup file (SETUP) and the help files (HLP extensions) must be placed in the system drive. It is suggested that EDIT, SETEDIT, SETUP if present, and all the help files be copied to a disk containing an operating system.

2.2 Terminal Configuration

A setup file called SETUP (with extension EDT) must be created before the editor may be used. On some systems, this file may be supplied. If so, then this section may be skipped.

The SETEDIT utility is used to define the characteristics of your particular terminal. The SETEDIT utility contains built in tables for the most widely used terminals. If you are using one of these terminals, then defining the terminal characteristics is a simple matter of selecting the proper terminal from a menu.

The following steps will guide you through the creation of the editor setup file.

- 1) Type SETEDIT <enter> to execute the utility
- 2) A menu will be displayed followed by a prompt to make a selection. Type T <enter> to define the terminal characteristics.
- 3) A list of terminals will be displayed, each preceded by a number. Type <enter> to view the remainder of the built-in terminal types. You will then be prompted to select a terminal. If your terminal is listed, then type in the correct number and proceed to step 5.

Otherwise, you must select either CUSTOM or SPECIAL. If your terminal is memory mapped video, then SPECIAL should be selected. Memory mapped video terminals will require some assembly language drivers to be written. See the appendix. If SPECIAL is selected, then proceed to step 4.

If you have an RS232 terminal, then CUSTOM should be selected. You will then be prompted for the type of cursor addressing. You must specify either binary or ASCII. The cursor is positioned to a particular location on the screen by specifying a row and column number. Some terminals use a single character to specify the row or column number. This is binary addressing. Other terminals expect a sequence of ASCII digits to specify the row or column. This is ASCII addressing. Normally, the 0 row or column position is not addressed with a 0 value. The next prompt asks for an offset value corresponding to row or column 0.

The next prompt is for what comes first, the row or column address? You will then be prompted for the character sequence preceding the row/column address. Next come two prompts for the character sequence between the row/column addresses and following the row/column addresses. If your terminal does not require any such sequence, simply type <enter> to these two prompts. Proceed to step 4.

- 4) You will be prompted for information about your terminal's characteristics and will need your terminal manual to answer the questions. The first two prompts are for the HEIGHT and WIDTH of your terminal. The height is the number of lines on the screen. The width is the number of characters on a line. The next sequence of prompts ask "does your terminal have this function?". If you answer yes, then you will be prompted to enter the character sequence to perform that particular function. See the example in the appendix. The following is a list of the functions which the editor supports.

clear to end of screen	- clear the screen from the current cursor position to the end of the screen.
clear to end of line	- clear the line from the current cursor position to the end of line.
insert line	- insert a blank line at the current cursor position.
delete line	- delete the line at the current cursor position.
delete character	- delete the character at the current cursor position.
enter insert mode	- cause the terminal to insert all subsequent characters at the current cursor position.
exit insert mode	- cause the terminal to stop inserting.
scroll 1 line down	- shift each line on the screen down by one line.
insert 1 character	- insert a character at the current cursor position.
scroll 1 line up	- shift each line on the screen up by one line.

- 5) The main menu is now redisplayed. Select option W to write a binary setup file and you will be prompted for a file name. Type the name SETUP followed by the extension EDT and press the <enter> key. The file will be written to disk. A drive specifier may also be included as part of the file name to place the file on a specific drive.
- 6) The main menu is once again displayed. Type E <enter> to exit the program. The setup file has been created and the editor may now be used.

2.3 Executing the Editor

The editor may be executed in several different ways. When creating a new file, simply type EDIT <enter>. The editor will configure itself with the default setup file and display *EOB at the top left corner of the screen indicating an empty text buffer. You may then insert one or more blank lines and start entering text.

The second way of executing the editor is to type EDIT FILENAME <enter>, where FILENAME is the name of some pre-existing ASCII text file. A drive specifier may also be included in the file name. The editor will configure itself with the default setup file and then load the first 100 lines from the specified file. The message LOADING... will be displayed at the bottom of the screen. Once the first 100 lines have been loaded, the editor will display a screen full of text starting with the first line and position the cursor at the top left corner of the screen. You may then begin editing.

The third way of executing the editor is to specify two file names in the form EDIT FILENAME1 FILENAME2 <enter>, where FILENAME1 is the name of the file to edit and FILENAME2 is the name of a setup file for the editor to use in configuration. As noted earlier, the editor by default loads the setup file named SETUP (with the extension EDT).

By specifying the setup file on the command line, the editor may be forced to use a setup file of some other name. This is convenient if you want the editor to default to different states, depending on the type of editing being performed. Drive specifiers may be included in either file name.

2.4 Basic Editor Commands

All the editor commands are explained in the following chapter. This section explains how some of the basic editor commands are mapped to the keyboard.

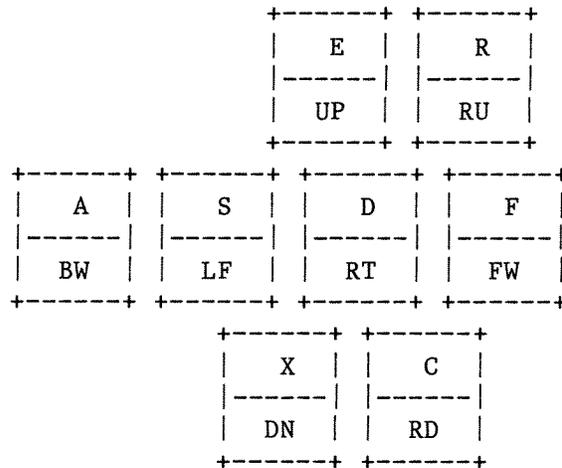
The editor has a large number of commands. Some of the commands are frequently used while others are used only occasionally. The commands which are used most often have been mapped to the keyboard. These commands may be executed while in compose mode by typing control characters. The remainder of the commands must be executed by entering command mode and typing a two character mnemonic. Command mode is entered by typing CTRL Z. Command mode allows you to execute one command and then the editor returns to compose mode. While in command mode, typing CTRL Z will abort command mode and return the editor to compose mode.

When creating a new file, the editor starts out with an empty text buffer. The symbol *EOB appears at the upper left corner of the screen. Before entering text, one or more blank lines must be inserted into the buffer. The insert line command has been mapped to CTRL N. Each time CTRL N is typed, a blank line is inserted into the buffer. Once the buffer contains lines, you may begin entering text. The <enter> key will cause the cursor to go to the beginning of the next line.

The editor defaults to overwrite mode. In overwrite mode, the editor will write directly over the character under the cursor. If there is text to the right of the cursor, the text will be changed as you type. The other mode is insert mode. There are two commands that cause the editor to enter insert mode. When CTRL V is typed, the editor temporarily enters insert mode. When characters are typed, they are inserted at the current cursor position. All characters to the right of the cursor will shift one character to the right each time a character is typed. When any editor command is executed, such as <enter>, the editor goes back into overwrite mode. The editor may be permanently placed in insert mode by entering command mode and typing IM <enter>. It may be placed back in overwrite mode by entering command mode and typing IM <enter> once again. This command toggles the editor from overwrite mode to insert mode and vice versa.

When the editor is in permanent insert mode, the <enter> key will insert carriage control. If the <enter> key is typed at the end of a line of text, the editor will insert a blank line following the current line and place the cursor at the beginning of the blank line. If the <enter> key is typed while in the middle of a line of text, the line will be split with the characters to the right of the cursor being placed on the inserted line. Insert mode is most useful when creating new text. It prevents the need for inserting blank lines in the text buffer before entering the text. When the buffer is empty, the <enter> key may be typed to insert one blank line and a new line is inserted automatically each time <enter> is typed thereafter.

The most frequently used editor commands are the cursor movement commands. These commands have been positionally mapped on the left side of the keyboard. The basic cursor movement commands are cursor right (RT), cursor left (LF), cursor up (UP), and cursor down (DN). These commands are mapped to the D,S,E, and X keys respectively. For example, typing CTRL D will cause the cursor to move one character to the right. Moving the cursor by word is a frequently used command. The (FW) command will move the cursor forward one word, while the (BW) command moves the cursor back one word. These commands have been mapped to the F and A keys respectively. The roll up command (RU) will scroll the text toward the beginning of the text buffer while the roll down command (RD) scrolls the text toward the end of the buffer. The number of lines scrolled is defaulted to 3 lines less than the size of the screen. These commands have been mapped to the R and C keys respectively. The following diagram illustrates the positional mapping used for these commands.



As noted earlier, command mode (CM) is mapped to Z and insert character mode (IC) is mapped to V. The delete character command (DC), which deletes the character under the cursor, is mapped to G. The delete word command (DW), which deletes the current word under the cursor, is mapped to T. The delete line command (DL), which deletes the current line under the cursor, is mapped to Y. Since CTRL H is typically associated with back space, it has been mapped to perform the cursor left command, same as CTRL S.

In addition to the single control character mapping, some commands have been mapped to a two character sequence. CTRL Q is used as a prefix for the commands labeled inside parentheses in the next diagram. After typing CTRL Q, these commands may be executed by typing the single character alone, or by typing CTRL <character>. For example, you could type CTRL Q, R or CTRL Q, CTRL R.

The top of buffer command (TP) causes the screen window to be moved to the top of the text buffer. The bottom of buffer command (BB) causes the screen window to be moved to the bottom of the text buffer. The beginning of line command (BL) causes the cursor to move to the leftmost column on the screen.

The end of line command (EL) causes the cursor to move one character past the right most character on the line. The center line command (CL) causes the current line to be centered on the screen. The home command (HM) causes the cursor to be positioned at the top left corner of the screen. The editor maintains a buffer for storing the last deleted line. The undelete line command (UL) will insert the last deleted line at the current cursor position.

```

+-----+
|  Q  |
+-----+
|prefix|
+-----+

+-----+ +-----+ +-----+ +-----+
|  E  | |  R  | |  T  | |  Y  |
+-----+ +-----+ +-----+ +-----+
| UP  | |RU(TP)| |  DW  | |DL(UL)|
+-----+ +-----+ +-----+ +-----+

+-----+ +-----+ +-----+ +-----+ +-----+ +-----+
|  A  | |  S  | |  D  | |  F  | |  G  | |  H  |
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+
| BW  | |LF(BL)| |RT(EL)| |FW(CL)| |  DC  | |LF(HM)|
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+

+-----+ +-----+ +-----+ +-----+
|  Z  | |  X  | |  C  | |  V  |
+-----+ +-----+ +-----+ +-----+
| CM  | |  DN  | |RD(BB)| |  IC  |
+-----+ +-----+ +-----+ +-----+

```

Other frequently used commands have been mapped to the right side of the keyboard. Since CTRL I generates the same character as the tab key, it has been mapped to the tab command (TB). Tabs are defaulted to 4 spaces. The back tab command (BT) has also been mapped to I. Back tab is executed when the CTRL Q prefix is used (ie. CTRL Q, I or CTRL Q, CTRL I).

The find next string command (FN) which searches for the next occurrence of a string has been mapped to J. Before executing FN, the find string command (FS), which defines a string and then searches for it, should be used. The FS command has been mapped to J with the Q prefix. The replace next string command (RN), which replaces the next occurrence of one string with another, has been mapped to L. Before executing RN, the replace string command (RS), which defines a string to search for and another string to use as replacement, should be used. The RS command has been mapped to L with the Q prefix.

The split line command (SP), which causes the current line to be split into two lines at the cursor position, has been mapped to O. The merge line command (MG), which causes the line following the cursor line to be merged with the cursor line, has been mapped to P. The delete to end of line command (DE), which deletes the current line from the cursor to the end of the line, has been mapped to K. The duplicate line command (DU), which duplicates the line above the cursor starting at the current cursor column, has been mapped to U.

As noted earlier, the insert line command (IL) has been mapped to N. CTRL M generates the same character as <enter> and is therefore equivalent. The mnemonic used for <enter> is (NL) which stands for next line.

```

+-----+
|   Q   |
|-----|
|prefix|
+-----+

+-----+ +-----+ +-----+ +-----+
|   U   | |   I   | |   O   | |   P   |
|-----| |-----| |-----| |-----|
|  DU  | |TB(BT)| |  SP  | |  MG  |
+-----+ +-----+ +-----+ +-----+

+-----+ +-----+ +-----+
|   J   | |   K   | |   L   |
|-----| |-----| |-----|
|FN(FS)| |  DE  | |RN(RS)|
+-----+ +-----+ +-----+

+-----+ +-----+
|   N   | |   M   |
|-----| |-----|
|  IL  | |  NL  |
+-----+ +-----+

```

There are several commands related to block operations. These commands have been mapped mnemonically, rather than positionally. (ie. the commands are mapped to keys that correspond to the first letter of the command mnemonic) The block commands must be prefixed by CTRL B. Simply type CTRL B, CTRL <key> or CTRL B, <key> where <key> is the key to which the specific command has been mapped. For example, CTRL B, M executes the mark command.

Any block operation must have a defined block of text to which the operation is applied. The mark command (MK) places an invisible mark at the line containing the cursor. Any block operation will be applied to the block of text between the marked line and the line currently containing the cursor. All block operations, except UR and LR occur on line boundaries. (ie. the column position of the cursor has no effect)

The three fundamental block operations are copy block (CB), insert block (IB), and delete block (DB). The copy block command copies the text within the marked region (ie. the text between the marked line and the line containing the cursor) to a temporary file. The insert block command inserts the text in the temporary file into the text buffer prior to the the line containing the cursor. The delete block command deletes the text within the marked region.

The print block command (PR) prints the text in the marked region. A line printer must be connected to use this command. The fill command (FI) requires two parameters, the left and right margins (column numbers). The fill command rearranges the text within the marked region so that it fits within the specified margins. The justify command (JF) also requires the same two parameters. The justify command rearranges the text on each line within the marked region so that the text aligns at both margins. This will cause extra blanks to be placed between words.

The upper case and lower case commands are the only block commands that operate on character boundaries rather than line boundaries. The upper case command (UR) makes all characters within the marked region upper case characters. The lower case command (LR) makes all characters within the marked region lower case characters.

There are two other very useful commands associated with the marked line and the line containing the cursor. The swap command (SW) swaps the marked line and the line containing the cursor. The cursor line becomes the new marked line and the cursor is positioned to the previously marked line. This command is useful in moving back and forth between two positions in the text buffer. The go to mark command (GM) causes the cursor to be positioned to the marked line.

2.5 Editor Help Files

The files named with HLP extensions are editor help files. These files contain information on editor commands. There are help files on the following subjects: (HELP, KEY, and CMD). These files may be viewed if you forget how to execute a particular command.

HELP displays information about the other two help files. KEY shows how the commands are mapped to keys. CMD lists all the editor commands in alphabetic order.

The help files may be viewed by typing CTRL Z to enter command mode and then typing HELP <enter>. When prompted for the subject, type in one of the above subjects followed by the <enter> key. If the <enter> key is typed without specifying a subject, the HELP file will be displayed. The help files must be on the system drive.

When viewing a help file, you may scroll downward towards the end of the file by typing CTRL C. To scroll back towards the beginning of the file, type CTRL R. To resume the edit session, type CTRL Z.

2.6 Swapping Disks

Sometime during an edit session, you may need to access files which are not on any of the disks currently in the drives. It is possible to swap disks during an edit session for such situations. On some systems (eg. CP/M), the swap disk editor command (SD) must be executed each time the disks are swapped. Type CTRL Z to enter command mode and then type SD <enter>.

When the editor is executed and has finished loading the setup file, the EDIT command file and the setup file are no longer needed during an edit session. The other files involved include the original file being edited and the editor work files. These files may be swapped as long as the following rules are followed.

- 1) The original file must be swapped back before an APPEND operation.
- 2) The main work file must be swapped back before a WRITE operation.
- 3) The original file and main work file must be swapped back before an EXIT or SAVE operation.

2.7 Exiting the Editor

There are two commands that can be used to terminate an edit session. Both are executed from command mode. Type CTRL Z to enter command mode. Angle brackets <> will appear at the lower left corner of the screen.

The EXIT command (EX) should be used if you wish to save the text in the buffer to a disk file. The EXIT command requires one parameter, the name of the file to which the buffer will be written. Simply type EX <enter>. The editor will prompt you to enter the name of a file, <EXIT>FILE:.. If creating a new file, a file name must be entered. The file name can include a drive specifier to force the file to a specific drive. Otherwise the file will be written to the default drive. If editing an existing file, you may enter a file name or simply press the the <enter> key. If no file name is entered, the buffer is written to the file specified when the editor was executed.

Before writing the buffer to the file, the editor prompts you with <EXIT>BACKUP:.. You can answer this prompt by typing either Y for yes or N for no. If Y <enter> is typed, the editor creates a backup file by renaming the original file with the extension BAK. This only has an effect if editing an existing file. If you answer the prompt with N <enter>, no backup file is created. Simply pressing the <enter> key to this prompt is equivalent to typing Y <enter>.

The QUIT command (QT) should be used if you wish to exit the edit session without saving the text buffer. The QUIT command requires one parameter, whether or not you really want to terminate the edit session. Simply type QT <enter>. The editor will prompt you with <QUIT>REALLY?. Type Y <enter> if you really wish to terminate the edit session without saving the buffer. Type N <enter> to resume the edit session.

Both the EXIT and QUIT commands return control to the operating system. If you wish to edit several different files at one time, the EXIT/ and QUIT/ commands should be used. These commands perform the same functions as EXIT and QUIT except that the editor remains loaded. Simply type EX/ <enter> or QT/ <enter> to execute these commands. After terminating the current edit session, the editor displays the *EOB symbol at the top left corner of the screen. You can then create a new file or edit an existing file. The APPEND command can be used to load in an existing file. Type CTRL Z to enter command

mode and type AP <enter>. The editor prompts for the number of lines you wish to append, <APPEND>LINES:. Enter the number of lines you wish to load from a file. The next prompt, <APPEND>FILE:, asks for the name of the file.

2.8 Sample Edit Session

The following steps show how a new file is created using the editor. Then the editor is used to edit the previously created file. Make sure the editor setup file and help files are on the system drive before beginning.

1. Type EDIT <enter>
2. When *EOB appears at the top left corner of the screen, type CTRL Z to enter command mode. Type IM <enter> to enter permanent insert mode. Then press the <enter> key.
3. Now simply type in the text. When you press the <enter> key, a blank line is inserted and the cursor is positioned to the beginning of the next line.
4. Several commands may be used to move around in the text to make changes or corrections. The previous section explained the commands which are mapped to keys. These commands are executed by holding down the CTRL key while pressing one of the alphabetic keys. For example, CTRL S moves the cursor left one character. Some of the commands must be prefixed by CTRL Q. For example, typing CTRL Q and then the S character will cause the cursor to go to the beginning of the current line.
5. If you forget how the commands are mapped to keys, type CTRL Z. Angle brackets will appear at the bottom left corner of the screen. Type HELP KEY <enter>. The screen will display help information about how the commands are mapped to keys. Type CTRL C to move forward in the help file. Type CTRL R to move backward. Type CTRL Z to resume the edit session.

6. In permanent insert mode, every time a character is typed it is inserted at the current cursor position. When the <enter> key is pressed, all text to the right of the cursor is moved to the next line. If you would rather that this did not occur, type CTRL Z to enter command mode and type IM <enter> to terminate permanent insert mode. Then when a character is typed, it is written over the character under the cursor rather than being inserted. The <enter> key then merely positions the cursor to the beginning of the next line. The editor will not permit the cursor to be moved pass the *EOB symbol. Type CTRL N to insert more blank lines when the *EOB symbol is reached.
7. Once you have finished entering the text, type CTRL Z to enter command mode. Type EXIT <enter>. You will be prompted with <EXIT>FILE:. Type in a valid file name. Drive numbers may be used as part of the file name. You will then be prompted with <EXIT>BACKUP? Simply press the <enter> key. Your file will be saved and the editor will exit to the operating system.
8. If you wish to modify the file, type EDIT followed by the filename used in step 7. The editor will load in the first 100 lines of the file. If fewer lines than this were in the file, the whole file will be loaded. Use the editor commands to move around in the text, making modifications. If not all lines were loaded into the text buffer, type CTRL Z to enter command mode and type APPEND 100 <enter>. One hundred more lines will be loaded from the file into the text buffer. If the buffer becomes full, (indicated by MEMORY EXHAUSTED message at the bottom of the screen) type CTRL Z to enter command mode. Type WRITE 100 <enter> and the first 100 lines in the buffer will be written to the editor work file. The lines written may not be edited again during the current edit session. After freeing buffer space with the WRITE command, more lines may be appended into the text buffer. When finished making changes, type CTRL Z to enter command mode.
9. If you wish to save your changes type EXIT <enter>. You may simply press the <enter> key to answer the following two prompts, <EXIT>FILE: and <EXIT>BACKUP?. The editor will save your changes to the file specified in step 8. However, before doing so, it will rename the original file created in step 7 to become a backup file. The same prefix of the file name is used with the extension BAK to represent that it is a backup file. The editor by default creates this backup file. To prevent its creation, you must type N to the <EXIT>BACKUP? prompt.
10. If you wish not to save your changes, type QUIT <enter>. When prompted with <QUIT>REALLY?, type Y <enter>. The editor will exit to the operating system and the file created in step 7 will be left unchanged.

Chapter 3

Editor Commands

The previous chapter explained only those commands which are internally mapped to the keyboard. The mapped commands may be executed by typing the appropriate control characters. This chapter explains all the editor commands except for a few special setup file commands. All these commands may be executed from command mode.

CTRL Z causes the editor to enter command mode. While in command mode, CTRL H may be used to backspace and correct typing errors. Commands are executed when the <enter> key is pressed. CTRL Z may be used prior to pressing the <enter> key to abort command mode and reenter compose mode.

All commands, some of which require parameters, have an associated two character mnemonic. In addition, the commands which require parameters have command names which may alternatively be used in place of the mnemonic. For example, the find string command requires a string parameter. The find string command may be executed from command mode by typing either FS or FIND, followed by the <enter> key. Abbreviations are also accepted. For example, simply typing F <enter> will execute the FIND command.

3.1 Command Parameters

When executing commands which require parameters, you may specify the parameters after the command name or you may simply type the command name without specifying the parameters. If the parameters are not specified, the editor will prompt for the required parameters. This is the case for key mapped commands which require parameters (eg. the FIND string command). The prompt will contain the long form of the command name inside angle brackets, followed by the parameter being requested. For example, typing FS <enter> while in command mode will result in the prompt <FIND>STRING:. You must then type in the string.

There are three types of parameters that are used for commands. Integer parameters are required for commands such as FILL (FI) and JUSTIFY (JF). With these commands, you must specify the columns to use as the left and right margins. For example, FILL 10 70 or JUSTIFY 5 75 might be used. When specifying integer parameters, blanks must be used to separate the individual parameters. Some commands have parameters which require a yes/no answer. These may be answered by typing YES or NO or by simply typing Y or N. Both upper and lower case are accepted. The third type of parameter is a string.

There are multiple ways of specifying string parameters. They may be quoted using either single or double quotes, or they may be unquoted. For example, either FIND 'ABC' or FIND "ABC" or FIND ABC could be used to locate the string ABC.

When specifying multiple string parameters on the command line, all but the last string parameter must be quoted. If the editor is expecting a string parameter and the next parameter is a non-quoted string, it will treat all characters to the end of the command line as part of the string. This may or may not be what was intended. For example, the replace string command takes two string parameters, a string to search for and one to use as replacement. This command could be executed in the following ways.

(1) RS 'ABC' BCD or (2) RS 'ABC' 'BCD' or (3) RS ABC BCD

Examples 1 and 2 are equivalent and would execute as intended. However, in example 3, the editor would use ABC BCD as a single string and then prompt for the next string.

In some circumstances, the quoted string is different from the unquoted string. The editor uses all characters in an unquoted string as they appear. However, it gives special meaning to certain characters in a quoted string. The # symbol is used to signify that a two character hex digit follows. The editor converts such 3 character sequences into a single character. For example, '#41' is converted to the single character A. The = symbol is used to signify that a two character command mnemonic follows. The editor converts this 3 character sequence into an internal command code. For example, '=RS' is converted to the internal editor code for the replace string command. The ^ symbol is used to represent the CTRL key. When this symbol is encountered, the editor converts the next printable character into the corresponding non-printable control character. For example, '^Q' is converted to the single ASCII character generated by CTRL Q. It is equivalent to '#11' (hexadecimal 11). See the ASCII character chart in the appendix.

If one of these special symbols is needed as a character in a quoted string, the symbol must appear twice. For example, '==' is equivalent to the single character =. The sequence '^ ^' however represents CTRL ^. Use '#5E' to represent the ^ character. Representation of the quote character itself within a string is handled in the same manner. For example, the string ''' represents a single quote character. An alternative is to use double quotes around a string that requires the single quote character, ""

In certain situations, such as defining a macro command (discussed in the following chapter), it may be necessary to use a quoted string within another quoted string. The editor accepts both single and double quotes as string delimiters. The case where you need a string within a string may be handled by using double quotes to delimit the outermost string, and single quotes to delimit the inner strings (eg. "RS 'cba' 'abc'").

3.2 Cursor Positioning Commands

The cursor positioning commands are commands which when executed result in the cursor being positioned to a different location within the text buffer.

3.2.1 New Line [NL]

The NL command moves the cursor to the beginning of the next line in the buffer. If in insert mode, the NL command moves all characters to the right of the cursor to the next line.

3.2.2 Right [RT]

The RT command moves the cursor one character to the right.

3.2.3 Left [LF]

The LF command moves the cursor one character to the left.

3.2.4 Up [UP]

The UP command moves the cursor one line towards the top of buffer.

3.2.5 Down [DN]

The DN command moves the cursor one line towards the bottom of the buffer.

3.2.6 Tab [TB]

The TB command moves the cursor right to the next tab stop. If in permanent insert mode, the tab command moves all characters to the right of the cursor to the next tab stop.

3.2.7 Back Tab [BT]

The BT command moves the cursor left to the next tab stop.

3.2.8 Forward Word [FW]

The FW command moves the cursor right to the beginning of the next word. A word is a sequence of non-blank characters.

3.2.9 Backward Word [BW]

The BW command moves the cursor left to the beginning of the next word. A word is a sequence of non-blank characters.

3.2.10 End of Line [EL]

The EL command moves the cursor one character past the last character on the line.

3.2.11 Beginning of Line [BL]

The BL command moves the cursor to the left edge of the screen.

3.2.12 Home [HM]

The HM command moves the cursor to the top left corner of the screen.

3.2.13 Roll Up [RU]

The RU command moves the cursor towards the top of the buffer. The number of lines moved is set by the ROLL command. The default is 4 lines less than the height of the screen

3.2.14 Roll Down [RD]

The RD command moves the cursor towards the bottom of the buffer. The number of lines moved is set by the ROLL command. The default is 4 lines less than the height of the screen

3.2.15 Top of Buffer [TP]

The TP command moves the cursor to the first line in the buffer.

3.2.16 Bottom of Buffer [BB]

The BB command moves the cursor to the last line in the buffer.

3.2.17 Go to Mark [GM]

The GM command moves the cursor to the marked line. The MK command is used to mark a line.

3.2.18 Swap [SW]

The SW command swaps the cursor and the marked line. The cursor is moved to the marked line and the marked line becomes the line containing the cursor at the time the command is executed. The MK command is used to mark a line.

3.2.19 Set Row [SR or ROW]

The SR command requires a parameter. The parameter is a row number of the screen. The SR command then moves the cursor to the specified row. For example, SR 0 <enter> moves the cursor to the top line of the screen.

3.2.20 Set Column [SC or COL]

The SC command requires a parameter. The parameter is a column number of the screen. The SC command then moves the cursor to the specified column. For example, SC 0 <enter> moves the cursor to the left edge of the screen.

3.2.21 Position [PO or POSITION]

The PO command requires two parameters. The first parameter is the row number of the screen. The second parameter is the column number of the screen. The PO command then moves the cursor to the specified row and column of the screen. For example, PO 0 0 <enter> moves the cursor to the top left corner of the screen.

3.2.22 Minus [MI or -]

The MI command requires a parameter. The parameter is the number of lines to move the cursor. The MI command then moves the cursor the specified number of lines towards the top of the buffer.

3.2.23 Plus [PL or +]

The PL command requires a parameter. The parameter is the number of lines to move the cursor. The PL command then moves the cursor the specified number of lines towards the bottom of the buffer.

3.2.24 Show Line(SL or SHOWLINE]

The SL command requires a parameter. The parameter is the number of a line in the buffer. The first line in the buffer is number 1. The SL command then moves the cursor to the specified line. All cursor movement commands except for the SL command apply only to the text buffer. The SL command will also accept a line number corresponding to a line in the file being edited. When editing a large file, the SL command can be used to quickly get a specific line loaded into the buffer. If the buffer cannot hold all the lines up to the specified line, the prior lines are written to the work file and the specified line becomes the first line in the buffer.

3.2.25 Horizontal Scroll [HS or HSCROLL]

The HS command requires a parameter. The parameter is the column that is positioned at the left edge of the screen. By default, column 1 is positioned at the left edge of the screen. The HS command is provided for terminals that have a width of less than 80 characters. The HS command can be used to scroll the screen left and right.

3.3 Inserting Text

The following commands are used to insert characters or lines into the text buffer.

3.3.1 Insert Mode [IM]

The IM command toggles the editor in and out of insert mode. The default is overwrite mode. In overwrite mode, the character under the cursor is replaced by the typed character. The new line command NL moves the cursor to the beginning of the next line. The tab command TB moves the cursor to the next tab stop. In insert mode, the editor inserts characters. All characters to the right of the cursor are shifted right as characters are typed. The new line command NL inserts a blank line and positions the cursor to the beginning of the blank line. If the NL command is executed in the middle of a line, all characters to the right of the cursor are moved to the next line. The tab command TB inserts blanks to position the character under the cursor at the next tab stop.

3.3.2 Insert Character [IC]

The IC command temporarily causes the editor to insert characters. The IC command is used while in overwrite mode to insert characters in the middle of a line. The editor stops inserting characters when a non-printable character is typed.

3.3.3 Insert Line [IL]

The IL command inserts a blank line into the buffer just prior to the line containing the cursor.

3.3.4 Undelete Line [UL]

The UL command inserts the last line deleted by the DL command just prior to the line containing the cursor.

3.3.5 Quote [QU]

The QU command causes the editor to enter the next character typed at the current cursor position. This command can be used to enter a non-printable character such as a control character into the buffer. The QU command prevents the editor from treating the next character typed as a command.

3.3.6 Quote String [QS or QUOTE]

The QS command requires a parameter. The parameter is a string of characters which the editor enters into the buffer starting at the position of the cursor. The QS command can be used to enter a string of non-printable characters into the buffer.

3.4 Deleting Text

The following commands are used to delete characters or lines from the text buffer.

3.4.1 Delete Character [DC]

The DC command deletes the character under the cursor.

3.4.2 Rub Out [RB]

The RB command deletes the character to the left of the cursor.

3.4.3 Delete Word [DW]

The DW command deletes the word under the cursor. A word is a sequence of non-blank characters. If the cursor is over a blank character, then the DW command deletes all surrounding blanks.

3.4.4 Delete Line [DL]

The DL command deletes the line under the cursor. The undelete line command UL can be used to restore a deleted line.

3.4.5 Delete to End [DE]

The DE command deletes all the characters from the cursor to the end of the line.

3.5 String Search and Replace

The following commands allow you to search for a specified character string within the text buffer. The editor maintains two string buffers. One is the find string buffer FSTRING and the other is the replace string buffer RSTRING. All of the following commands require parameters that define values for one or both of these string buffers.

3.5.1 Find String [FS or FIND]

The FS command requires one parameter. The parameter is the string that will be searched for within the text buffer. The FS command then begins searching for the string starting at the first character to the right of the cursor. If the string is found, the cursor is positioned over the first character in the string. The message STRING NOT FOUND is displayed at the bottom of the screen if the string is not found.

3.5.2 Replace String [RS or REPLACE]

The RS command requires two parameters. The first parameter is the string that will be searched for within the text buffer. The second parameter is the string that will be used as a replacement. The RS command then begins searching for the string starting at the first character to the right of the cursor. If the string is found, it is replaced by the string in the replace string buffer. The message STRING NOT FOUND is displayed at the bottom of the screen if the string is not found.

3.5.3 Find Next [FN]

The FN command searches for the next occurrence of the string in the find string buffer. The search begins with the character to the right of the cursor. If the string is found, the cursor is positioned over the first character in the string. The message STRING NOT FOUND is displayed at the bottom of the screen if the string is not found.

3.5.4 Replace Next [RN]

The RN command searches for the next occurrence of the string in the find string buffer and replaces it with the string in the replace string buffer. The search begins with the character to the right of the cursor. If the string is found, it is replaced. The message STRING NOT FOUND is displayed at the bottom of the screen if the string is not found.

3.5.5 Replace Global [RG or REPGLOB]

The RG command requires two parameters. The first parameter is the string that will be searched for within the text buffer. The second parameter is the string that will be used as a replacement. The RG command works the same as the RS command except that it replaces all occurrences of the string from the cursor to the end of buffer, instead of just the first one encountered.

3.6 Block Commands

The following commands operate on a block of text. A block is the text between and including the marked line and the line containing the cursor. The mark command is used to mark a line. Some of the commands use a block buffer to store or retrieve a block of text. All of the commands except for LR and UR operate on line boundaries. The LR and UR commands operate on character boundaries.

3.6.1 Mark [MK]

The MK command places an invisible mark on the line under the cursor.

3.6.2 Copy Block [CB]

The CB command copies the block of text from the marked line to the cursor into the block buffer.

3.6.3 Insert Block [IB]

The IB command inserts the block buffer into the text buffer just prior to the line containing the cursor.

3.6.4 Delete Block [DB]

The DB command deletes the block of text from the marked line to the line containing the cursor.

3.6.5 Lower Case [LR]

The LR command converts all characters from the marked character to the character under the cursor to lower case.

3.6.6 Upper Case [UR]

The UR command converts all characters from the marked character to the character under the cursor to upper case.

3.6.7 Print [PR]

The PR command prints the text from the marked line to the line containing the cursor. A line printer must be connected before executing this command.

3.6.8 Fill [FI or FILL]

The FI command requires two parameters. The first parameter specifies the column to use as the left margin. The second parameter specifies the column to use as the right margin. The FI command then fills the text from the marked line to the line containing the cursor so that all characters fit between the defined margins.

3.6.9 Justify [JF or JUSTIFY]

The JF command is equivalent to the FI command except that the text is also justified. All the text from the marked line to the line containing the cursor is aligned at the defined margins.

3.6.10 Extract [XT or EXTRACT]

The XT command requires one parameter. The parameter is the name of a file. The XT command then writes all the text from the marked line to the line containing the cursor to the specified file.

3.7 File Commands

The file commands are operations involving disk files.

3.7.1 Help [HP or HELP]

The HP command requires one parameter. The parameter is the name of a help file (excluding extension). The supplied help files are HELP, KEY, and CMD. The HELP file contains information about the other two help files. The KEY file illustrates how the editor commands are mapped to keys. The CMD file contains a complete list of editor commands in alphabetic format. On some systems (eg. CP/M), the help files must be on the system drive. The HP command then displays the help file. While viewing a help file, the RU command (CTRL R) and the RD command (CTRL C) can be used to scroll back and forth in the file. The CM command (CTRL Z) terminates the HP command.

3.7.2 Directory [DI or DIR]

The DI command requires one parameter. The parameter is a drive specifier, usually a number or a letter, depending on the operating system. Simply pressing the <enter> key selects the default drive. The DI command then displays a list of the files on the disk in the specified drive.

3.7.3 Show File [SF or SHOWFILE]

The SF command requires one parameter. The parameter is the name of a file. The SF command then displays the specified file. While viewing the file, several commands can be used to move around in the file. The RU (CTRL R) and RD (CTRL C) commands can be used to scroll back and forth in the file. The UP (CTRL E) and DN (CTRL X) commands will scroll the file back and forth by one line. You can type a line number followed by the <enter> key to display a specific line number at the top of the screen. Typing - or + and a number and then pressing the <enter> key will scroll the specified number of lines toward the top or bottom of the file. The CM command (CTRL Z) terminates the SF command.

3.7.4 Insert File [IF or INSFILE]

The IF command requires 3 parameters. The first parameter is the name of a file. The second parameter is a starting line number in the file. The third parameter is the number of lines to insert. The IF command then inserts the specified number of lines beginning with the specified starting line number in the specified file. The lines are inserted into the buffer just prior to the line containing the cursor. If the file contains fewer lines than specified, all lines to the end of the file are inserted.

3.7.5 Delete File [DF or DELFILE]

The DF command requires a parameter. The parameter is the name of the file to delete. The DF command then deletes the file. The DF command is useful in deleting unneeded files to make more disk space available during an edit session. The DI command can be used to display the directory of a disk to locate the file(s) you wish to delete.

3.7.6 Save [SV or SAVE]

The SV command is equivalent to the EX command except that the edit session is resumed once the text buffer is saved to a file. The SV command can be used to periodically save the text buffer during an edit session. The message DONE is displayed at the bottom of the screen when the SV command finishes execution.

3.7.7 Append [AP or APPEND]

The AP command requires either one or two parameters. The first parameter is the number of lines. The second parameter is the name of the file. The AP command then appends the specified number of lines from the file to the end of the text buffer. The second parameter is not required if editing an existing file and there are lines remaining in the file that have not been appended to the buffer. Specifying a file when executing the editor is equivalent to not specifying a file and then using the AP command to append 100 lines from a file. If the file contains more than 100 lines, the AP command must be used to load the remaining lines into the buffer. The AP command does not prompt for a file name until all lines have been loaded from the file being appended. Once all the lines have been loaded from the file, the next execution of the AP command requires a file to be specified.

3.7.8 Write [WR or WRITE]

The WR command requires one parameter. The parameter is the number of lines to write to the work file. The WR command then writes the specified number of lines from the text buffer to the work file. The lines are appended to the end of the work file. The writing begins with the first line in the buffer. Once the lines have been written to the work file, they are deleted from the buffer. The WR command is used to free space in the text buffer so that more lines can be appended from a file. The AP and WR commands are often used together when editing files too large to fit in the text buffer. These commands allow you to page through the file.

3.8 Setting and Clearing Tab Stops

The following commands are used to set or clear tab stops. By default, tab stops are set at four character intervals.

3.8.1 Delete Tabs [DT]

The DT command clears all tab stops.

3.8.2 Set Tab [ST]

The ST command sets a tab stop at the current cursor position.

3.8.3 Clear Tab [CT]

The CT command clears the tab stop at the current cursor position.

3.8.4 Tab Stops [TS or TABS]

The TS command has two forms. The first form requires a single integer parameter. This form sets a tab stop at the specified character intervals. For example, TABS 4 sets a tab stop at 4 character intervals. The second form allows tab stops to be set at specific columns. This form requires a sequence of one or more column numbers separated by commas. The column sequence must be separated from the command name by an = symbol. For example, TABS = 5,20,40,60 sets a tab stop at each specified column.

3.9 Miscellaneous

3.9.1 Command Mode [CM]

The CM command places the editor in command mode. Angle brackets <> are displayed at the bottom left corner of the screen and the editor waits for a command to be executed. Commands are executed by typing a two character command mnemonic. The CTRL H key can be used to back space if typing errors are made. When the <enter> key is pressed, the command is executed.

3.9.2 Duplicate [DU]

The DU command copies the characters above and to the right of the cursor onto the line containing the cursor. The DU command can be used to make multiple copies of a particular line.

3.9.3 Merge [MG]

The MG command appends the line below the cursor to the end of the line containing the cursor. The merged line is limited to 80 characters. Excess characters are not merged. They are left on the line below.

3.9.4 Split [SP]

The SP command splits a line into two lines. When the SP command is executed, a blank line is inserted below the line containing the cursor and all characters to the right of the cursor are moved to the blank line.

3.9.5 Center Line [CL]

The CL command centers on the screen the line identified by the cursor.

3.9.6 Auto Indent [AI]

The AI command toggles the auto indent feature on and off. When auto indent is off, the cursor is positioned at the left edge of the screen when the <enter> key is pressed. When auto indent is on, the cursor is positioned under the first non-blank character of the line above when the <enter> key is pressed. The default is off.

3.9.7 Line Numbers [LN]

The LN command causes the editor to display a line number for each line in the text buffer. The LN command toggles line numbers on and off. The default is off.

3.9.8 Memory [MM]

The MM command displays the amount of unused memory remaining in the text buffer. When the MM command is executed, the editor displays two numbers at the bottom of the screen. The first number is the number of characters that the buffer has room left to store. The second number is the percent of total buffer space that remains unused.

3.9.9 Refresh [RF]

The RF command causes the editor to redisplay the screen. This command may be useful to determine whether or not non-printable characters are in the buffer. If the display behaves improperly, then the buffer contains a non-printable character that the terminal is interpreting as a command.

3.9.10 Tabify [TF]

The TF command toggles tab compression on and off. When tab compression is off, the file will not contain any tab characters when the EX command is executed. When tab compression is on, the file will contain tab characters for each leading sequence of 8 blanks when the EX command is executed. The default is on.

3.9.11 Swap Disk [SD]

The SD command must be executed before changing disks on some operating systems [eg. CP/M).

3.9.12 Roll [RL or ROLL]

The RL command requires one parameter. The parameter is the number of lines that the cursor is moved when the RU or RD command is executed. The default is 4 lines less than the height of the screen.

3.10 The EDIT Command [ED]

The editor has a command that allows a second file to be edited without terminating the current edit session. When the editor is executed from the operating system it starts out at level 1. This is the only level ever used unless the ED command is executed. This command causes the editor to go to level 2. When level 2 is entered, the editor clears the text buffer and essentially begins a new edit session. The editor state remains unchanged except for the fact that a new file is being edited. The level 1 edit session is preserved with its current state at the time level 2 is entered. When level 2 is terminated by either the QT or EX commands, level 1 is reentered in its preserved state. The two levels are independent. [ie. the level 2 edit does not effect anything in the level 1 edit or vice versa).

The ED command requires five parameters. The first parameter is the name of the file to edit. If a file is specified, the first 100 lines are loaded into the text buffer. If the <enter> key is typed for the file prompt, the level two edit will start with an empty buffer. The next four parameters define an edit window. They define the top row, bottom row, left column, and right column of the screen. The editor will use only this window to display text. The rest of the screen will be left undisturbed. This is a useful feature if you need to view a few lines of text from the level 1 edit while editing in level 2. If the specified left and right column parameters define a window too narrow to display all the text horizontally, the horizontal scroll command HS may be used to scroll the text in and out of the window. The window parameter prompts may be answered by typing the <enter> key. When the <enter> key is typed, the default value used is the screen boundary.

3.11 Terminating an Edit Session

There are four commands that can be used to terminate an edit session.

3.11.1 Exit [EX or EXIT]

The EX command requires two parameters. The first parameter is the name of the file. This can be answered by simply pressing the <enter> key if editing an existing file. The second parameter is a Y or N answer to whether or not to create a backup file. Simply pressing the <enter> key is equivalent to typing Y <enter>. The EX command then writes the text buffer to the file and exits to the operating system. The backup file is created only if the file already exists. The backup is made by renaming the original file with a BAK extension. The work file is then named as the original. The backup reflects the file contents just prior to the last edit session.

3.11.2 Exit/ [E/ or EXIT/]

The E/ command is equivalent to the EX command except the editor remains loaded with an empty text buffer. The E/ command should be used when editing multiple files at one time.

3.11.3 QUIT [QT or QUIT]

The QT command requires one parameter. The parameter is a Y or N answer to whether or not you really wish to terminate the edit session without saving the text buffer. If Y is answered, the QT command exits to the operating system. Otherwise the edit session is resumed.

3.11.4 QUIT/ [Q/ or QUIT/]

The Q/ command is equivalent to the QT command except the editor remains loaded with an empty text buffer. The Q/ command should be used if you simply wish to delete the text buffer.

Chapter 4

Changing Editor Characteristics

The editor has several commands that change specific characteristics of the editor. This chapter explains the two commands that allow you to customize the editor. One command allows you to change the way the keyboard is mapped to the editor commands. The other allows you to define your own editor commands. A single command may be constructed from the set of built in commands. This is called a macro command because it combines more than one of the built in commands to form a single more powerful command.

4.1 Translating Keys to Commands

The editor is supplied with selected commands internally mapped to the keyboard. It is possible to change this mapping partially or completely to suit personal preference. The TRANS command will allow you to map any keyboard generated character sequence to any editor command. The character sequence consists of one or more ASCII characters, printable or non-printable. It is best to start the sequence with a non-printable character (eg. the ESC character or a control character). If a printable character starts the sequence, you will no longer be able to enter that character as text. The editor treats all character sequences defined by the TRANS command as a single entity. As each character is received from the keyboard, it is checked to see if it begins a sequence mapped to an editor command. If not, then the character is entered as text if it is printable or discarded if it is non-printable. If the character does begin a defined sequence, the appropriate command is executed for the case of a single character sequence. For multiple character sequences, subsequent characters are used to determine the appropriate command. (ie. they are not entered as text) Only after the sequence traces to a specific command or an invalid sequence does the editor revert back to entering printable characters as text.

There are three special symbols which may be used in quoted strings. (#, ^, and =). The editor gives special meaning to these symbols. If the symbol itself is to be a character in the string, it must appear twice. For example, '##' would be the single character #.

Any ASCII character may be represented in a quoted string. The non-printable characters may be represented by a three character sequence beginning with the # symbol. The two characters following the symbol must be a valid hexadecimal value. The ASCII chart in the appendix shows the

hexadecimal value for each character in the ASCII character set.

The ^ symbol is used to represent the control key, CTRL. For example, CTRL Q may be represented in a string as '^Q'. This is equivalent to the string '#11' since CTRL Q has the ASCII value of hexadecimal 11. As you can see, '^Q' is a little more readable than '#11'. The character sequence CTRL Q, f, Z is represented as '^QfZ'. The sequence CTRL W, CTRL X is represented as '^W^X'. The escape character ESC may be represented as '^['.

The = symbol is used to signify that a two character editor command mnemonic follows. For example, the string '=UP' represents the cursor up command (UP). The string '=FS' represents the find string command (FS).

4.1.1 Translate [TR or TRANS]

The TR command requires two string parameters. The first parameter is a sequence of one or more keys. The second parameter is a single character or command mnemonic. The TR command stores the key sequence and its associated character or command in a table. The TR command can be used to cause a key to generate a particular character. For example, TR '?' '\ ' causes the editor to translate the ? key to the \ character. However, the TR command is usually used to translate a non-printable key to an editor command. For example, TR '^W' '=FW' translates CTRL W to the FW command. The command TR '^Q^W' '=BW' translates CTRL Q, CTRL W to the BW command.

While in command mode, rather than type ^ followed by a character, the actual control character may be typed. The editor will echo the two character representation. For example, if CTRL R is typed, the editor will echo ^R to the command line. When the ESC key is typed, the editor echos ^[to the command line.

4.2 Defining Macro Commands

It is possible to translate a key sequence to more than one character or editor command. The DEFINE command allows a sequence of one or more keys to be translated to a sequence of one or more characters and/or commands.

4.2.1 Define Macro [DM or DEFINE]

The DM command requires two string parameters. The first parameter is a sequence of one or more keys. The second parameter is a sequence of one or more characters and/or editor commands. The DM command stores the key sequence and its associated character/command sequence in a table. When the defined key sequence is typed, the editor uses the associated character/command string as input. The editor treats the string as if you were typing it from the keyboard. Characters are entered into the text buffer and commands are executed. Essentially anything that can be done manually

from the keyboard can be defined in this string and executed automatically.

A simple illustration of the use of the define command is to map a key to a word which is often typed. The following example maps the word Program to the key sequence ESC P (ie. the escape key followed by upper case P). Then typing ESC P is equivalent to typing the word Program.

```
DEFINE '^[P' 'Program'
```

Commands may be specified in several ways. One way is to specify the command mnemonic preceded by the = symbol. If the command requires parameters, the parameters should immediately follow the mnemonic. Each parameter must be followed by the NL command as a parameter terminator. By default, the NL command is mapped to the <enter> key which is equivalent to CTRL M. Therefore, either ^M or =NL may be used as a parameter terminator. The following macro will move the cursor forward by sentence. The example maps the macro to ESC S. When executed, the find string command positions the cursor over the next period in the text. Then the RT command is used to handle cases where the period is the last character on a line. The FW command then causes the cursor to be positioned to the beginning of the next word.

```
DEFINE '^[S' '=FS.=NL=RT=FW'
```

First the FS command is executed which causes the editor to prompt for the <FIND>STRING: parameter. The editor then uses the sequence of characters up to the terminator as the find string parameter, in this case a period. Once the parameter is input, the FS command is executed. When finished, the next input received is the cursor right and forward word commands. After the FW command is executed, control returns to the keyboard.

The following example illustrates another way of defining the macro to move forward by sentence. The keys that the commands are mapped to can be used in place of the command mnemonics.

```
DEFINE '^[S' '^Q^J.^M^D^F'
```

The difference between the two macros is that the first definition will not change if a key is remapped. The second definition will. For example, if TR '^D' '=DC' is executed, the second macro definition then deletes the period because ^D is now delete character instead of cursor right.

Any defined key sequence can be used in defining a macro. This means that one macro may reference another defined macro. Since strings may not cross line boundaries, this provides a way of building macros longer than one line. The following example illustrates a macro definition which refers to another defined macro.

Suppose you wish to define a macro to capitalize the first character of the current word under the cursor. First define a macro which positions the cursor to the beginning of the current word. The BW command will do this.

However, if the cursor is already on the first character of a word, the back word command will position the cursor to the beginning of the previous word. To prevent this from occurring, position the cursor right one character before executing the BW command.

```
DEFINE '^[B' '=RT=BW'
```

Now you can define a macro which capitalizes the character under the cursor. The UR block command can be used to do this. First the MK command is used to mark the current cursor position. Then the UR command capitalizes the character under the cursor.

```
DEFINE '^[C' '=MK=UR'
```

A macro can now be defined which uses both of the previously defined macros.

```
DEFINE '^W' '^[B^[C'
```

CTRL W can then be used to capitalize the current word under the cursor. This particular macro is short enough to have been defined as a single macro. However, it does illustrate how one macro may use another defined macro.

The DM command can be used to create lots of other useful editor commands. For example, the DM command can be used to define commands to page through a large file. The WR and AP commands must be used to page through files that are too large to fit in the text buffer. After defining the following macro, 100 lines at a time are paged from the file into the text buffer when CTRL Q, CTRL P is typed.

```
DEFINE '^Q^P' '=WR100=NL=AP100=NL'
```

4.2.2 Undefine Macro [UM or UNDEFINE]

The maximum number of macros that may be defined at any one time is 64. If you wish to change the macro defined to a particular key sequence, you may simply use the DM command to define a new macro to that particular sequence. However, the memory required to store the previously defined macro will not be recovered. To recover the memory used by the old macro, the undefine macro command UM should be used.

The UM command requires one parameter. The parameter is a string corresponding to a key sequence. The UM command then deletes the defined key sequence and associated command string from the table.

Chapter 5

Editor Setup Files

The editor must use a setup file at a minimum to determine the terminal characteristics. In addition, the setup file may be used to customize the editor by setting specific editor states, mapping commands to keys, and defining new commands. Several setup files may be created to allow the editor to be configured differently each time it is executed. Then simply specify the appropriate setup file when the editor is executed. You might use one setup file for programming and another one for word processing. If you use more than one programming language, you might have a separate setup file for each language.

There are some normal editor commands which are allowed in setup files and some special commands which can be used only in setup files. Among the special commands are four terminal defining commands (HEIGHT, WIDTH, TERMINAL, and CURSOR). These are the commands which the SETEDIT utility outputs to a setup file to define terminal characteristics. When creating a setup file, you may exclude the terminal characteristics. After creating a text format setup file using the editor, the SETEDIT utility may be used to read it, merge it with the terminal information, and then write the combined information to a binary and/or text format file. The terminal information may be merged by selecting the proper terminal from the menu or by reading a previously created setup file containing the terminal information. A binary setup file containing terminal information should be present on your master disk if your computer has a known terminal type. This setup file is named SETUP with an EDT extension. If not present, the SETEDIT utility must be used to create the setup file. Remember, the editor requires binary setup files. However, the text format setup file is useful if you wish to make modifications.

The commands in setup files are limited to one command per line. The semicolon (;) may be used as a comment specifier. If a semicolon is encountered in a setup file, the remaining text on that line is treated as a comment. This of course does not apply to semicolons which appear inside quoted strings. The commands may be placed in any order within the setup file. The only requirement is that a key sequence must be defined before it is referenced by the START or DEFINE commands.

5.1 Normal Commands

This section lists the normal editor commands which may be used in a setup file. In setup files, these commands must be specified using the command name. The mnemonic is not allowed. Otherwise, the form is the same as described in chapters 3 and 4.

5.1.1 TABS

The TABS command may be used to change the default tab setting. The default is TABS 4. Both forms of the TABS command may be used. See the TS command in chapter 3.

5.1.2 ROLL

The ROLL command may be used to change the default setting for the number of lines scrolled by the RU and RD commands. The default roll size is four less than the screen height. See the RL command in chapter 3.

5.1.3 AUTOINDENT

The AUTOINDENT command can be used to turn on the auto indent feature from a setup file. The default is off. See the AI command in chapter 3.

5.1.4 TRANS

The TRANS command can be used to change the default mapping of commands to keys. A complete remapping may be performed or the default mapping may be slightly modified. See the TR command in chapter 4.

5.1.5 DEFINE

The DEFINE command may be used to define macro commands formed from the built in commands. If a macro command references a key sequence defined by the TRANS command or by another DEFINE command, the referenced sequence must be defined on a previous line. There is a limit of 64 macro definitions. See the DM command in chapter 4.

5.2 Special Commands

The special commands are commands which may only be used in a setup file. The last four commands in this section define terminal characteristics. These commands are automatically created by the SETEDIT utility and therefore may be included in the setup file through the use of this utility.

5.2.1 INIT

The INIT command may be used to send a string of characters to the terminal when the setup file is loaded. The command requires one parameter which is a quoted string. The string may contain either printable or non-printable characters. Printable characters may be sent to identify the setup file being used for the current edit session. Non-printable characters that the terminal interprets as commands may also be used to set some desired terminal characteristic.

Example: INIT 'Pascal Setup File'

5.2.2 EXIT

The EXIT command is identical to the INIT command except the string is not sent to the terminal until the editor is exited. There can only be one EXIT command in a setup file.

Example: EXIT 'Edit Finished'

5.2.3 START

The START command specifies a string of commands that are executed immediately after the file to be edited has been loaded. This command can be used to execute editor commands that are not allowed in the setup file. The following example causes the editor to start out in insert mode with tab compression turned off. There can only be one START command in a setup file.

Example: START '=IM=TF'

5.2.4 CMD

The CMD command provides a way of giving names to the built in editor commands. The commands which do not require parameters have only a two character mnemonic. The CMD command may be used to define a longer name for the command. In some cases, a longer name may be easier to remember than the two character mnemonic. It requires two parameters. The first is the name. The second is a string containing an editor mnemonic. The following example assigns the name MARK to the mnemonic MK. This will allow the mark command to be executed by either the full name or the two character mnemonic.

Example: CMD MARK '=MK'

The following four commands describe the terminal characteristics. The SETEDIT utility may be used to create these commands in a setup file.

5.2.5 HEIGHT

The HEIGHT command requires a single integer parameter that defines the number of lines on the screen.

Example: HEIGHT 24

5.2.6 WIDTH

The WIDTH command requires a single integer parameter that defines the character width of the screen.

Example: WIDTH 80

5.2.7 TERMINAL

The TERMINAL command defines the features of the terminal. It requires two parameters. The first is a name that identifies a terminal function. The second parameter is a string containing the character sequence required by the terminal to perform that specific function. The editor makes use of most of the smart features included in many of the latest terminals. The following features are supported.

CLEAR - clear screen
 CLREOS - clear to end of screen
 CLREOL - clear to end of line
 INSLINE - insert line
 DELLINE - delete line
 DELCHAR - delete character
 INSMODE - enter insert mode
 NOINS - exit insert mode
 RSCROLL - scroll the screen 1 line down
 (reverse linefeed with cursor at top of screen)
 SCROLL - scroll the screen 1 line up
 (linefeed with cursor at bottom of screen)
 INSONE - insert one character

Other parameters of the terminal command specify how cursor addressing is performed.

CURSOR - specifies the character sequence which precedes the row and column. This parameter is used if the terminal does not require character sequences between and following the row and column address.

CURSOR1 - specifies the character sequence which precedes the row and column. This parameter is used if the terminal requires character sequences between and following the row and column address.

CURSOR2 - specifies the character sequence which must appear between the row and column.

CURSOR3 - specifies the character sequence which follows the row/column address.

COFFSET - specifies the offset for addressing the first row or column on the screen.

Example: `TERMINAL CLEAR '^[Y'`

5.2.8 CURSOR

The CURSOR command requires one parameter that describes the algorithm used to address the cursor. The following is a list of the possible cursor addressing methods. The SETEDIT utility will generate one of these addressing methods.

ROWCOL, ANSII, COLROW, BINARY, ASCII, SPECIAL

Example: `CURSOR ROWCOL`

5.3 Sample Setup Files

```

;*****
;*
;*          TRS80 MODEL II/12/16 SETUP FILE          *
;*          (for Lifeboat CP/M)                      *
;******
;----- terminal characteristics -----
;          (Created by the SETEDIT utility)
;
TERMINAL CLEAR      '^[:'          ;clear screen
TERMINAL CLREOS    '^[y'          ;clear to end of screen
TERMINAL CLREOL    '^[T'          ;clear to end of line
TERMINAL INSLINE   '^[E'          ;insert line
TERMINAL DELLINE   '^[R'          ;delete line
TERMINAL DELCHAR   '^[W'          ;delete character
TERMINAL CURSOR    '^[='          ;address cursor
TERMINAL INSONE    '^[Q'          ;insert one character
TERMINAL SCROLL    '^[J'          ;scroll 1 line up
CURSOR ROWCOL      ;row/column cursor addressing
HEIGHT            24              ;number of lines/screen
WIDTH             80              ;number of characters/line
;----- end of terminal definition -----
;
;          (User Created Section)
;
INIT 'TRS80 MODEL 12 CONFIGURATION';send message to terminal
EXIT 'EDIT SESSION FINISHED'      ;send message to terminal
;
;KEY TRANSLATIONS
;-----
TRANS '^\' '=LF'          ;left arrow mapped to cursor left
TRANS '^]' '=RT'          ;right arrow mapped to cursor right
TRANS '^^\ '=UP'          ;up arrow mapped to cursor up
TRANS '^_ '=DN'          ;down arrow mapped to cursor down
;
;EDITOR STATE CONFIGURATION
;-----
START '=TF=IM'          ;tab compression off, insert mode on
TABS 8                  ;set tabs every 8 spaces
AUTOINDENT              ;turn on auto-indent
ROLL 23                 ;set scrolling to screen height - 1
;
;DEFINE COMMAND NAMES
;-----
CMD MARK '=MK'          ;mark
CMD TOP  '=TP'          ;top of buffer

```

```

CMD BOTTOM '=BB'           ;bottom of buffer
CMD INDENT '=AI'          ;auto indent
;
;DEFINE MACROS
;-----
DEFINE '^[@' '=IL=IL=IL=IL=IL' ;ESC @-insert 5 blank lines
DEFINE '^[N' '^[@^@'         ;ESC N-insert 10 blank lines
DEFINE '^[1' 'PROGRAM'       ;ESC 1-type PROGRAM
DEFINE '^[2' 'CONST'         ;ESC 2-type CONST
DEFINE '^[3' 'TYPE'          ;ESC 3-type TYPE
DEFINE '^[4' 'VAR'           ;ESC 4-type VAR
DEFINE '^[5' 'PROCEDURE '    ;ESC 5-type PROCEDURE
DEFINE '^[6' 'FUNCTION '     ;ESC 6-type FUNCTION
DEFINE '^[7' 'BEGIN '       ;ESC 7-type BEGIN
DEFINE '^[8' 'END'           ;ESC 8-type END
DEFINE '^[D' '^[2^M^[3^M^[4' ;ESC D-type declarations
DEFINE '^[B' '^[7^M^[8'     ;ESC B-type body
DEFINE '^[F' '^[6^M^[D^M^[B;' ;ESC F-type function
DEFINE '^[P' '^[5^M^[D^M^[B;' ;ESC P-type procedure
;
; ESC A puts a program shell on the screen
;     there must be blank lines in the buffer
;     if the editor is not in insert mode
;
DEFINE '^[A' '^M=HM^[1^M^[D^M^M^[F^M^M^[P^M^M^[B.=HM=FW '
;
;Make the ESC <character> work with lower case
;
DEFINE '^[n' '^[N'
DEFINE '^[d' '^[D'
DEFINE '^[b' '^[B'
DEFINE '^[f' '^[F'
DEFINE '^[p' '^[P'
DEFINE '^[a' '^[A'
; end of setup file

```

```

;*****
;*          TRS-80 MODEL 4 SAMPLE SETUP FILE          *
;*          (for TRSDOS 6)                            *
;*  Among other things, this setup file maps the arrow *
;*  keys to cursor movement commands.                 *
;*  This setup file maps some commands to the clear  *
;*  and break keys.  In the explanations below, the  *
;*  clear key is represented as <CLR> and the break  *
;*  key as <BRK>.  When executing commands mapped with *
;*  <CLR>, the clear key should be held down.  When  *
;*  executing commands mapped with <BRK>, the break  *
;*  key should be pressed and released.              *
;*  The keys which are mapped to commands are commented *
;*  in the form:   ;key                               --> command *
;*  Note:                                              *
;*  The editor's internal mapping of keys to commands *
;*  remains valid for all keys which are not explicitly *
;*  remapped by this setup file.                    *
;*****
;-----terminal definition -----
;          (This section was created by SETEDIT)
;
TERMINAL CLEAR      '^ \ ^ '          ;clear screen
TERMINAL CLREOS    '^ | '           ;clear to end of screen
TERMINAL CLREOL    '^ ~ '           ;clear to end of line
TERMINAL SCROLL    '^ J '           ;scroll 1 line up
CURSOR SPECIAL     ;special cursor addressing
HEIGHT            24                ;number of lines/screen
WIDTH             80                ;number of characters/line
;----- end of terminal definition -----
;
;          (Customization Section)
;
;  The following two commands send strings of characters
;  to the terminal.  ^N in the INIT string insures that
;  the Model 4 cursor is turned on.
;
INIT '^NReading Setup File'         ;send message at start
EXIT 'Edit Session Finished'        ;send message at end
;
;KEY TRANSLATIONS
;-----
;  The following key translations redefine how editor
;  commands are mapped to the Model 4 keyboard.
;  Appendix B of the Model 4 Disk System Owners's
;  Manual has a keyboard diagram which shows the
;  characters generated by each key.
;
;-----
;  The following key translations map the arrow keys
;  to cursor movement commands.  The left arrow key
;  generates ^H which is internally mapped to =LF.
TRANS '^I' '=RT'          ;right arrow          --> cursor right

```

```

TRANS '^J'  '=DN'      ;down arrow      --> cursor down
TRANS '^K'  '=UP'      ;up arrow      --> cursor up
TRANS '#8A' '=RD'      ;<CLR> down arrow --> roll down
TRANS '#8B' '=RU'      ;<CLR> up arrow  --> roll up
TRANS '#88' '=BT'      ;<CLR> left arrow --> back tab
TRANS '#89' '=TB'      ;<CLR> right arrow --> tab
;
; The following key translations map various commands
; mnemonically using the clear key as a control key.
; Use <CLR> N for insert line.
;
TRANS '#C6' '=FN'      ;<CLR> F        --> find next
TRANS '#D2' '=RN'      ;<CLR> R        --> replace next
TRANS '#C3' '=DC'      ;<CLR> C        --> delete character
TRANS '#D7' '=DW'      ;<CLR> W        --> delete word
TRANS '#CC' '=DL'      ;<CLR> L        --> delete line
TRANS '#D5' '=UL'      ;<CLR> U        --> undelete line
TRANS '#C4' '=DU'      ;<CLR> D        --> duplicate line
TRANS '#C9' '=IC'      ;<CLR> I        --> insert character
;
; The following key translations map commands to the
; 3 function keys F1, F2, and F3. The shifted function
; keys are represented as <SFn>
;
TRANS '#81' '=CM'      ;<F1>          --> command mode
TRANS '#82' '=BW'      ;<F2>          --> backward by word
TRANS '#83' '=FW'      ;<F3>          --> forward by word
TRANS '#91' '=IM'      ;<SF1>         --> insert mode
TRANS '#92' '=SP'      ;<SF2>         --> split line
TRANS '#93' '=MG'      ;<SF3>         --> merge line
;
; The following key translations map commands to the
; keys using the break key <BRK> as a prefix.
;
TRANS '#80#0A' '=BB'   ;<BRK> down arrow --> bottom of buffer
TRANS '#80#0B' '=TP'   ;<BRK> up arrow  --> top of buffer
TRANS '#80#08' '=BL'   ;<BRK> left arrow --> beginning of line
TRANS '#80#09' '=EL'   ;<BRK> right arrow --> end of line
TRANS '#80L'   '=DE'   ;<BRK> L        --> delete to end of line
TRANS '#801'   '=DE'   ;<BRK> l        --> delete to end of line
;
;EDITOR STATE CONFIGURATION
;-----
; The following commands set default states for the
; editor.
;
START  '=TF'          ;tab compression off
TABS   3              ;set tabs every 3 spaces
AUTOINDENT          ;turn on auto-indent
ROLL   23            ;set scrolling to screen height - 1
;
;DEFINE LONG NAMES FOR THESE COMMANDS
;-----

```

```

; The following commands define long names which may
; be used while in command mode to execute these commands.
;
CMD MARK      '=MK'           ;MARK      is equivalent to MK
CMD INDENT    '=AI'           ;INDENT   is equivalent to AI
;
;DEFINE MACROS
;-----
; The following commands define macro's which map
; Pascal keywords to the numeric keys using clear as a
; control key.
;
DEFINE '#B1'  'PROGRAM '      ;<CLR> 1 --> PROGRAM
DEFINE '#B2'  'CONST '        ;<CLR> 2 --> CONST
DEFINE '#B3'  'TYPE '         ;<CLR> 3 --> TYPE
DEFINE '#B4'  'VAR '           ;<CLR> 4 --> VAR
DEFINE '#B5'  'PROCEDURE '     ;<CLR> 5 --> PROCEDURE
DEFINE '#B6'  'FUNCTION '      ;<CLR> 6 --> FUNCTION
DEFINE '#B7'  'BEGIN '         ;<CLR> 7 --> BEGIN
DEFINE '#B8'  'END '           ;<CLR> 8 --> END
;
; The following commands define macros's which use
; the macro's defined above to create Pascal program shells.
; The next line command (=NL) is internally mapped to ^M
; which is generated by the <enter> key. ^M is used in
; place of =NL in the definitions below.
;
DEFINE '#80D' '#B2^M#B3^M#B4' ;<BRK> D --> declarations
DEFINE '#80B' '#B7^M#B8'      ;<BRK> B --> body
DEFINE '#80F' '#B6^M#80#44^M#80#42;' ;<BRK> F --> function
DEFINE '#80P' '#B5^M#80#44^M#80#42;' ;<BRK> P --> procedure
;
; The following macro definition defines
; <BRK> S to put a complete program shell on the screen
;
DEFINE '#80S' '=IM=BL^M^K#B1^M#80D^M^M#80P^M^M#80F^M^M#80B.=HM=FW =IM'
;
; The following definitions make the above macros work
; with lower case.
;
DEFINE '#80d' '#80D'          ;<BRK> d --> <BRK> D
DEFINE '#80b' '#80B'          ;<BRK> b --> <BRK> B
DEFINE '#80f' '#80F'          ;<BRK> f --> <BRK> F
DEFINE '#80p' '#80P'          ;<BRK> p --> <BRK> P
DEFINE '#80s' '#80S'          ;<BRK> s --> <BRK> S
;
; The following macros define keys which terminate an edit
; session. <BRK> E is defined to exit and save the file
; being edited but not save a backup file. <BRK> Q is
; defined to quit the edit without saving the file.
;
DEFINE '#80E' '=EX=NLN=NL'    ;<BRK> E --> exit
DEFINE '#80Q' '=QTY=NL'       ;<BRK> Q --> quit

```

```
DEFINE '#80e' '#80E'           ;<BRK> e --> <BRK> E
DEFINE '#80q' '#80Q'           ;<BRK> q --> <BRK> Q
;
; end of setup file
```


Appendix A

Custom Setup

A.1 Sample Terminal Setup

This is a sample execution of the SETEDIT utility using the CUSTOM terminal selection. The terminal used in the sample is the TELEVIDEO 925/950. Note that for steps 26 and 30, ^@ was used. This is the null character. Null characters are ignored by most terminals. They may therefore be used as fill characters in order to control timing. If a particular terminal function responds too slowly, null characters may be used to allow the terminal time to complete the function.

- 1) Type SETEDIT
- 2) Please make a selection: T <enter>
- 3) Please select a terminal or 0 to exit: 31 <enter>
- 4) Do you want to continue? Y <enter>
- 5) B=binary, A=ascii: B <enter>
- 6) Which is first, row or column (R,C): R <enter>
- 7) enter a decimal number (space=32): 32 <enter>
- 8) What characters come before the row number: ^= <enter>
- 9) What characters come between the row and column: <enter>
- 10) What characters come after the column number: <enter>
- 11) Does your terminal have clear screen ? Y <enter>
- 12) Sequence to perform it: ^= <enter>
- 13) Does your terminal have clear to end screen ? Y <enter>
- 14) Sequence to perform it: ^=Y <enter>
- 15) Does your terminal have clear to end of line? Y <enter>
- 16) Sequence to perform it: ^=T <enter>
- 17) Does your terminal have insert line ? Y <enter>
- 18) Sequence to perform it: ^=E <enter>
- 19) Does your terminal have delete line ? Y <enter>
- 20) Sequence to perform it: ^=R <enter>
- 21) Does your terminal have delete character ? Y <enter>
- 22) Sequence to perform it: ^=W <enter>
- 23) Does your terminal have enter insert mode ? N <enter>
- 24) Does your terminal have exit insert mode ? N <enter>
- 25) Does your terminal have scroll 1 line down ? Y <enter>
- 26) Sequence to perform it: ^=j^@^@^@ <enter>
- 27) Does your terminal have insert 1 character ? Y <enter>
- 28) Sequence to perform it: ^=Q <enter>
- 29) Does your terminal have scroll 1 line up ? Y <enter>
- 30) Sequence to perform it: ^=J^@ <enter>

- 31) Please make a selection: W <enter>
- 32) Enter name for binary setup file: SETUP.EDT
- 33) Please make a selection: E <enter>

A.2 Special Cursor Addressing

Many computer systems that use the CP/M operating system can be used with a variety of terminal types. This section shows how to create a screen driver for a terminal that does not use a character sequence to address the cursor.

The minimum requirement for executing the Blaise II editor is that the display device must have an addressable cursor. The custom cursor addressing option provided in the SETEDIT utility will allow the use of almost any terminal that allows the cursor to be directly addressed. For custom cursor addressing to work, the terminal must allow direct cursor addressing in response to a set of characters sent to it. This is normally some form of escape sequence with the cursor address embedded.

If your terminal (or memory mapped video) does not allow cursor addressing via a sequence of characters, then you must use the special cursor addressing option. To use special cursor addressing, execute the SETEDIT utility and select option 32 (special) for the terminal type. This will cause the editor to use a customer supplied assembly language routine to address the cursor. At this time, you can specify any escape sequences that your terminal supports.

You must write an assembly language routine to perform cursor addressing. When the cursor is to be moved, the editor will transfer control to address 1C21H with the row number contained in register H and the column number in register L. Your assembly language routine should use this information to perform whatever action is required to move the cursor and then return to the editor with a normal RET instruction. All of the registers are available and your routine may occupy up to 64 bytes of memory. The code should be assembled with an origin of 1C21H (hexadecimal) and a .HEX file generated. This module will be merged with the Blaise II editor in the next steps.

To merge the assembly language driver with the Blaise II editor, perform the following steps.

1. Load the editor with DDT by: DDT EDIT.COM
2. Use the I command in DDT to specify the .HEX file containing your driver
3. Use the R command to read the driver into memory
4. Type control-C to exit to CP/M
5. Save the new image with: SAVE 150 EDIT.COM

The following example shows a sample assembly language driver and the method for merging it with the editor. It assumes that a 16 line by 64 character memory mapped video board is installed in the system at address FC00H. The cursor address is stored as a direct memory address at location 40H

in memory, and that the most significant bit of the byte under the cursor is set to make the cursor byte inverse video.

```
;
; sample cursor addressing for 16x64 memory mapped video
;
MEMORY EQU 0FC00H
CURSOR EQU 040H
CSRBIT EQU 080H
;
;          ORG 01C21H
;
; on entry, the row is in H and the column is in L
;
    PUSH H
    LHLD CURSOR          ; deselect old cursor
    MOV A,M
    XRI CSRBIT
    MOV M,A
    POP D                ; get new cursor address
    MOV A,D
    RAR                  ; shift row into position
    RAR
    MOV D,A              ; save shifted row
    ANI 3                ; mask upper two bits
    MOV H,A
    MOV A,D              ; get low 2 bits of row
    ANI 0COH            ; mask
    ADD E                ; add column
    MOV L,A
    LXI D,MEMORY        ; add board offset
    DAD D
    SHLD CURSOR         ; save new cursor
    MOV A,M              ; set cursor byte to inverse
    XRI CSRBIT
    MOV M,A
    RET                  ; exit back to editor
END
```

If the above subroutine is contained in the file A:CURSOR.ASM, then it can be merged into the editor as shown below. The text following the semicolons (;) are comments and would not be entered.

```
A>ASM CURSOR ; assemble the source
CP/M ASSEMBLER - VER 2.0
1C42
000H USE FACTOR
END OF ASSEMBLY

A>DDT EDIT.COM ; load the editor with DDT
DDT VERS 2.2
NEXT PC
9700 0100
-ICURSOR.HEX ; setup to load
-R ; read the hex file
NEXT PC
9700 0100
-^C ; type control C to exit
A>SAVE 150 EDIT.COM ; save the new image
A>
```

Appendix B

Standard 7-bit ASCII Character Set

Decimal	Octal	Hex	Graphic	Name
0.	000	00	^@	NUL (used for padding) <null>
1.	001	01	^A	SOH (start of header)
2.	002	02	^B	STX (start of text)
3.	003	03	^C	ETX (end of text)
4.	004	04	^D	EOT (end of transmission)
5.	005	05	^E	ENQ (enquiry)
6.	006	06	^F	ACK (acknowledge)
7.	007	07	^G	BEL (bell or alarm)
8.	010	08	^H	BS (backspace) <bs>
9.	011	09	^I	HT (horizontal tab) <tab>
10.	012	0A	^J	LF (line feed) <lf>
11.	013	0B	^K	VT (vertical tab)
12.	014	0C	^L	FF (form feed, new page) <ff>
13.	015	0D	^M	CR (carriage return) <cr>
14.	016	0E	^N	SO (shift out)
15.	017	0F	^O	SI (shift in)
16.	020	10	^P	DLE (data link escape)
17.	021	11	^Q	DC1 (device control 1, XON)
18.	022	12	^R	DC2 (device control 2)
19.	023	13	^S	DC3 (device control 3, XOFF)
20.	024	14	^T	DC4 (device control 4)
21.	025	15	^U	NAK (negative acknowledge)
22.	026	16	^V	SYN (synchronous idle)
23.	027	17	^W	ETB (end transmission block)
24.	030	18	^X	CAN (cancel)
25.	031	19	^Y	EM (end of medium)
26.	032	1A	^Z	SUB (substitute)
27.	033	1B	^[ESCAPE (alter mode, SEL) <esc>
28.	034	1C	^\	FS (file separator)
29.	035	1D	^]	GS (group separator)
30.	036	1E	^^	RS (record separator)
31.	037	1F	^_	US (unit separator)
32.	040	20	" "	space or blank <sp>
33.	041	21	!	exclamation mark
34.	042	22	"	double quote
35.	043	23	#	number sign (hash mark)
36.	044	24	\$	dollar sign
37.	045	25	%	percent sign
38.	046	26	&	ampersand sign
39.	047	27	'	single quote (apostrophe)

40.	050	28	(left parenthesis
41.	051	29)	right parenthesis
42.	052	2A	*	asterisk (star)
43.	053	2B	+	plus sign
44.	054	2C	,	comma
45.	055	2D	-	minus sign (dash)
46.	056	2E	.	period (decimal point)
47.	057	2F	/	(right) slash
48.	060	30	0	numeral zero
49.	061	31	1	numeral one
50.	062	32	2	numeral two
51.	063	33	3	numeral three
52.	064	34	4	numeral four
53.	065	35	5	numeral five
54.	066	36	6	numeral six
55.	067	37	7	numeral seven
56.	070	38	8	numeral eight
57.	071	39	9	numeral nine
58.	072	3A	:	colon
59.	073	3B	;	semi-colon
60.	074	3C	<	less-than sign
61.	075	3D	=	equal sign
62.	076	3E	>	greater-than sign
63.	077	3F	?	question mark
64.	100	40	@	atsign
65.	101	41	A	upper-case letter ABLE
66.	102	42	B	upper-case letter BAKER
67.	103	43	C	upper-case letter CHARLIE
68.	104	44	D	upper-case letter DELTA
69.	105	45	E	upper-case letter ECHO
70.	106	46	F	upper-case letter FOXTROT
71.	107	47	G	upper-case letter GOLF
72.	110	48	H	upper-case letter HOTEL
73.	111	49	I	upper-case letter INDIA
74.	112	4A	J	upper-case letter JERICHO
75.	113	4B	K	upper-case letter KAPPA
76.	114	4C	L	upper-case letter LIMA
77.	115	4D	M	upper-case letter MIKE
78.	116	4E	N	upper-case letter NOVEMBER
79.	117	4F	O	upper-case letter OSCAR
80.	120	50	P	upper-case letter PAPPA
81.	121	51	Q	upper-case letter QUEBEC
82.	122	52	R	upper-case letter ROMEO
83.	123	53	S	upper-case letter SIERRA
84.	124	54	T	upper-case letter TANGO
85.	125	55	U	upper-case letter UNICORN
86.	126	56	V	upper-case letter VICTOR
87.	127	57	W	upper-case letter WHISKY
88.	130	58	X	upper-case letter XRAY
89.	131	59	Y	upper-case letter YANKEE
90.	132	5A	Z	upper-case letter ZEBRA
91.	133	5B	[left square bracket
92.	134	5C	\	left slash (backslash)
93.	135	5D]	right square bracket
94.	136	5E	^	uparrow (carat)

95.	137	5F	_	underscore
96.	140	60	`	(single) back quote
97.	141	61	a	lower-case letter able
98.	142	62	b	lower-case letter baker
99.	143	63	c	lower-case letter charlie
100.	144	64	d	lower-case letter delta
101.	145	65	e	lower-case letter echo
102.	146	66	f	lower-case letter foxtrot
103.	147	67	g	lower-case letter golf
104.	150	68	h	lower-case letter hotel
105.	151	69	i	lower-case letter india
106.	152	6A	j	lower-case letter jericho
107.	153	6B	k	lower-case letter kappa
108.	154	6C	l	lower-case letter lima
109.	155	6D	m	lower-case letter mike
110.	156	6E	n	lower-case letter november
111.	157	6F	o	lower-case letter oscar
112.	160	70	p	lower-case letter pappa
113.	161	71	q	lower-case letter quebec
114.	162	72	r	lower-case letter romeo
115.	163	73	s	lower-case letter sierra
116.	164	74	t	lower-case letter tango
117.	165	75	u	lower-case letter unicorn
118.	166	76	v	lower-case letter victor
119.	167	77	w	lower-case letter whisky
120.	170	78	x	lower-case letter xray
121.	171	79	y	lower-case letter yankee
122.	172	7A	z	lower-case letter zebra
123.	173	7B	{	left curly brace
124.	174	7C		vertical bar
125.	175	7D	}	right curly brace
126.	176	7E	~	tilde
127.	177	7F	<rubout>	DEL

Index

APPEND 5
Append [AP or APPEND] 33
Auto Indent [AI] 35
AUTOINDENT 44
Back Tab [TB] 24
Backward Word [BW] 24
Beginning of Line [BL] 24
Bottom of Buffer [BB] 25
buffer 5
Center Line [CL] 35
Clear Tab [CT] 34
CMD 46
command mode 7
Command Mode [CM] 35
compose mode 6
Copy Block [CB] 30
CURSOR 47
DEFINE 44
Define Macro [DM or DEFINE] 40
Delete Block [DB] 31
Delete Character [DC] 28
Delete File [DF or DELFILE] 33
Delete Line [DL] 28
Delete Tabs [DT] 34
Delete to End [DE] 28
Delete Word [DW] 28
Directory [DI or DIR] 32
Down [DN] 23
Duplicate [DU] 35
Edit [ED] 37
End of Line [EL] 24
EXIT 45
Exit [EX or EXIT] 37
EXIT/ 5
Exit/ [EX or EXIT/] 38
Extract [XT or EXTRACT] 32
file configuration 9
Fill [FI or FILL] 31
Find Next [FN] 29
Find String [FS or FIND] 29
Forward Word [FW] 24
Go to Mark [GM] 25
HEIGHT 46
help files 17
Home [HM] 24
Horizontal Scroll [HS or HSCROLL] 26
INIT 45
Insert Block [IB] 31

Insert Character [IC] 27
 Insert File [IF or INSFILE] 33
 Insert Line [IL] 27
 Insert Mode [IM] 27
 Justify [JF or JUSTIFY] 31
 Left [LF] 23
 Line Numbers [LN] 36
 Lower Case [LR] 31
 Mark [MK] 30
 Memory [MM] 36
 Merge [MG] 35
 Minus [MI or -] 26
 New Line [NL] 23
 Plus [PL or +] 26
 Position [PO or POSITION] 25
 Print [PR] 31
 Quit [QT or QUIT] 38
 Quit/ [Q/ or QUIT/] 38
 Quote [QU] 27
 Quote String [QS or QUOTE] 28
 Refresh [RF] 36
 Replace Global [RG or REPGLOB] 30
 Replace Next [RN] 30
 Replace String [RS or REPLACE] 29
 Right [RT] 23
 ROLL 44
 Roll [RL or ROLL] 36
 Roll Down [RD] 24
 Roll Up [RU] 24
 Rub Out [RB] 28
 Save [SV or SAVE] 33
 Set Column [SC or COL] 25
 Set Row [SR or ROW] 25
 Set Tab [ST] 34
 SETEDIT 4, 9
 SETUP 3
 setup files 3
 Show File [SF or SHOWFILE] 32
 Show Line [SL or SHOWLINE] 26
 Split [SP] 35
 START 45
 Swap [SW] 25
 Swap Disk [SD] 36
 Tab [TB] 23
 Tabify [TF] 36
 TABS 44
 Tabs [TS or TABS] 34
 TERMINAL 46
 terminal configuration 9
 Top of Buffer [TP] 25
 TRANS 44
 Translate [TR or TRANS] 40
 Undefine Macro [UM or UNDEFINE] 42
 Undelete Line [UL] 27
 Up [UP] 23
 Upper Case [UR] 31

WIDTH 46
work file 6
WRITE 5
Write [WR or WRITE] 34

Table of Contents

Chapter 1 System Description	1
1.1 System Utilities	1
1.1.1 EDIT	1
1.1.2 SETEDIT	1
1.1.3 CC	1
1.1.4 CCB	2
1.1.5 OPTIMIZE	2
1.1.6 CODEGEN	2
1.1.7 HEXTOBIN	3
1.1.8 RUNC	3
1.1.9 LINKLOAD	4
1.2 Files and Devices	4
Chapter 2 Using the System	7
2.1 Using the Compiler	7
2.1.1 Short Form	7
2.1.2 Long Form	8
2.1.3 Compiler Listing	9
2.2 Using the Run Utility	10
2.3 Using the Linking Loader	11
2.3.1 Load Command	11
2.3.2 Find Command	12
2.3.3 Symbols Command	12
2.3.4 Run Command	13
2.3.5 Build Command	13
2.3.6 Init Command	14
2.3.7 Exit Command	14
2.3.8 Error Messages	14
Chapter 3 Miscellaneous	17
3.1 Compiler Memory Constraints	17
3.2 Runtime Memory Usage	17
3.3 Accessing Arguments via Pointers	18
3.4 Size and Range of Basic Types	18
3.5 Generating EOF from the Keyboard	19
3.6 Linking Assembly Language	19
3.6.1 Using the XASM Assembler	19
3.6.2 Using Another Assembler	20

3.7 Patches	20
Chapter 4 Function Libraries	23
4.1 SYSTEM Library	24
4.1.1 read	24
4.1.2 write	24
4.2 CLIB, PRINTF, & SCANF Libraries	25
4.2.1 _initio	25
4.2.2 open	25
4.2.3 creat	26
4.2.4 lseek	27
4.2.5 close	28
4.2.6 unlink	29
4.3 TRSLIB Library	30
4.3.1 SVC	30
4.3.2 TIME	31
4.3.3 DATE	31
4.3.4 SOUND	31
4.3.5 CMDLINE	31
4.3.6 USER	32
4.3.7 CALL\$	32
4.3.8 \$MEMORY	33
4.3.9 HP\$ERROR	33
4.3.10 PEEK	33
4.3.11 POKE	34
4.3.12 INP	34
4.3.13 OUT	34
4.3.14 WRITECH	35
4.3.15 WRITESTRING	35
4.3.16 INKEY	35
4.3.17 GETKEY	36
4.3.18 FILE\$STATUS	36
4.3.19 IO\$ERROR	36
4.3.20 DELFILE	36
4.3.21 RENAME	37
4.3.22 SET\$ACNM	37
4.3.23 SETACNM	37
4.3.24 CLEARGRAPHICS	38
4.3.25 CLEARSCREEN	38
4.3.26 GOTOXY	38
4.3.27 NOBLANK	38
4.3.28 READCURSOR	39
4.3.29 RSETPOINT	39
4.3.30 SETPOINT	39
4.3.31 TESTPOINT	39
4.3.32 EXTMEM	40
4.4 RANDOM Library	42

4.4.1 OPENRAND	42
4.4.2 READRAND	43
4.4.3 WRITERAND	43
4.4.4 CLOSERAND	44
4.4.5 Notes and Error Codes	44
4.4.6 Example	44
4.5 STRINGS Library	46
4.5.1 LEN	46
4.5.2 LEFT\$	46
4.5.3 RIGHT\$	46
4.5.4 MID\$	46
4.5.5 STR\$	47
4.5.6 ENCODEI	47
4.5.7 ENCODER	47
4.5.8 ENCODED	47
4.5.9 DECODEI	48
4.5.10 DECODER	48
4.5.11 DECODED	48
4.5.12 CHARACTER	48
4.5.13 CMPSTR	48
4.5.14 CONC	49
4.5.15 CPYSTR	49
4.5.16 DELETE	49
4.5.17 FIND	49
4.5.18 INSERT	50
4.5.19 REPLACE	50

Chapter 1

System Description

1.1 System Utilities

The program development system consists of several utilities supplied as executable command files (CMD extensions).

1.1.1 EDIT

The Blaise II text editor is a customizable full screen text editor that is used to create programs. The word customizable refers to its ability to be configured by the user. Editor commands may be mapped to the keyboard to suit personal preference and macro commands may be defined. A macro command is a defined sequence of built-in editor commands that may be executed by pressing a single key.

1.1.2 SETEDIT

A setup file is a file that the editor uses each time it is executed. It contains a user defined configuration for the editor. The setedit utility is a program that is used to create and modify setup files.

1.1.3 CC

The C compiler is simply a program that translates C source programs into an intermediate language called p-code. The p-code is a low level language that resembles the assembly language for a stack oriented computer.

Once a program has been compiled, the object p-code program is stored as an object file (OBJ). The OBJ file may be executed directly or may be run through the advanced development package (ADP).

There is no limit to the size of program that can be compiled since a program may be composed of many functions which are split into separate source files. However, there is a limit to the size of an individual source file. The size of a C source file that may be compiled is dependent on the number of

symbols used in the source file and not necessarily the number of lines. Excessive use of macro definitions (#define) greatly reduces the size of file that can be compiled. The compiler should be able to compile a typical 200 line file of C source code. Minimizing the use of global variables allows larger files to be compiled.

1.1.4 CCB

An overlaid C compiler is provided on Disk3 for compiling larger programs. Since it must load overlays during the compilation process, it is slower than the non-overlaid compiler and should be used only if you do not have enough memory to compile a program. The overlay files (OVn extensions) must be present when using CCB.

1.1.5 OPTIMIZE

The optimizer is one of the two utilities described in the Advanced Development Package (ADP) Manual. It is optionally used to further process a p-code object file.

The OPTIMIZE utility inputs an OBJ file and outputs an optimized object file, OPT. The purpose of the optimizer is to remove statement redundancy in the translated object code. The result is a p-code object file that is approximately 10 to 30 percent smaller. The optimizer should be used where program size is important. The optimized p-code is an exceptionally compact representation of the C program.

1.1.6 CODEGEN

The code generator is the second of the two utilities described in the ADP Manual. The code generator inputs either an OBJ or OPT file and outputs a machine code object file, COD. The purpose of the code generator is to translate p-code instructions (which must be interpreted) to machine instructions (which the computer can execute directly). The result is a machine code object file that executes approximately 3 to 5 times faster than a p-code object file. The code generator should be used when execution speed is important.

A single p-code instruction will typically translate into 2 or 3 machine instructions. Therefore, a COD file will typically be 2 to 3 times larger than an OBJ file. This is not a problem for small programs but may be for large programs. The object code may become too large to fit into the available memory when the program is executed.

In a typical large program, 90% of the time is spent executing only 10% of the program. For large programs, selected functions that effect performance the most can be translated to machine instructions. The bulk of the program can remain p-code instructions. All three types of files, OBJ, OPT, and COD can be linked together to form a program. Therefore, large programs can be developed without sacrificing performance.

In C, extern (global) variables limit the size of program that can be handled by CODEGEN. The size problem usually occurs when using large arrays. The problem may be solved by simply placing the global variables in a separate file from the program. The program then declares the variables using the extern keyword and the two files are compiled separately. Once compiled, the file containing the program is then run through CODEGEN. Since the other file contains only data, it need not be run through CODEGEN. The Linking Loader is then used to link the two files together.

1.1.7 HEXTOBIN

The hex to binary utility on Disk3 converts hex object files (readable) to binary object files (non-readable). This utility can be used on any object file created by CC, CCB, OPTIMIZE, or CODEGEN. Translating hex files to binary reduces the size of the file by approximately 30% and results in faster loading by the RUNC and LINKLOAD utilities. When this utility is executed, it prompts for the name of the hex object file to use as input and the binary object file to use as output. The file extensions must be specified. As a convention to distinguish binary object files from hex object files, you may want to name binary files with the extension BIN.

1.1.8 RUNC

The RUNC utility loads and executes an OBJ, OPT, COD, or BIN object file. This utility provides a fast way to execute a compiled program. The standard C functions from SYSTEM/OBJ, CLIB/OBJ, PRINTF/OBJ, and SCANF/OBJ are built into RUNC.

The RUNC utility loads and executes a single object file. If the object file contains an unresolved reference (a call to a function not present in the file), then RUNC tries to open the file named RUNC/OBJ to resolve the references. This provides a mechanism for using RUNC even when performing separate compilations. Functions not present in RUNC/CMD or in the main object code file may be placed in the file named RUNC/OBJ. The supplied RUNC/OBJ file on disk C3 is an example. It contains all the functions from TRSLIB/OBJ, RANDOM/OBJ, and STRINGS/OBJ. If this file is present with RUNC/CMD, your program may also call a function in one of these libraries. The RUNC/OBJ file was created by appending the three libraries together to form a single file. This single file was then processed by the HEXTOBIN utility to make the file smaller and load faster.

1.1.9 LINKLOAD

The LINKLOAD utility may be used to load an OBJ, OPT, COD, or BIN file. Multiple object files (created by separate compilations) may be loaded. The LINKLOAD utility links the separate object files together as they are loaded. This utility does not contain the supplied external library functions. If a program uses one or more of the functions in an external library, the library object file must be loaded to link the function(s) to the program.

After the object files are loaded, the program may be executed or a command file may be built. A command file is a stand alone program that the operating system can load and execute.

1.2 Files and Devices

All the utilities accept file names in the syntax of the TRSDOS 6 operating system. TRSDOS 6 file names have up to four parts.

The main part is 1 to 8 characters long and identifies the file. This part must begin with a letter and may contain any alphanumeric characters.

The second part of the file name is an optional extension which is separated from the rest of the name by a slash (/). The extension may be 1 to 3 characters in length and is usually used to identify the type of the file (eg. /C for C source, /OBJ for object code, etc.).

Some of the utilities supply default file extensions. For example, when the C compiler is executed with a file specified on the command line, the compiler assumes that the extension is /C and it places the object code into a file with the same name but an extension of /OBJ. The RUNC command assumes that its input has the extension /OBJ unless otherwise specified.

OPTIMIZE by default takes its input from a file with an extension of /OBJ and writes its output to a file by the same name with an extension of /OPT. In a similar manner, CODEGEN uses /OBJ as a default input and /COD as the corresponding output.

The third part of the file name is an optional password. The password is a sequence of 1 to 8 alphanumeric characters, the first of which must be a letter. The password is used to limit access to a file.

The fourth part of the file name is the disk drive. This part is also optional. The drive number is separated from the rest of the file name by a colon (:). If a drive number is not specified, all available drives will be searched. The search begins with drive number 0 and continues until the file is found or there are no more drives to search. If a drive number is specified, only that drive is searched. Specifying a drive number will insure

that a file is placed on a specific drive and will also speed up file access time.

Example File Names:

```
DATABASE
ACCOUNT/DAT
SAMPLE2/OBJ:1
REPORT.POIU
SECRET/C.REWQ:1
HOMEWRK:2
```

All the system utilities accept device names in addition to file names. Device names are single characters preceded by a colon (:). The following devices are supported.

```
:C      the CRT (terminal screen)
:L      the Line Printer
:D      dummy device
```


Chapter 2

Using the System

This chapter describes how to use the C compiler (CC), the run utility (RUNC), and the linking loader (LINKLOAD). The EDIT Manual describes the use of the full screen text editor and the ADP Manual describes the use of the OPTIMIZE and CODEGEN utilities.

2.1 Using the Compiler

The compiler may be executed in two different ways.

2.1.1 Short Form

The short form requires the minimum amount of typing. It allows a program to be compiled using a single line command.

```
CC <stack-size> file-name
```

where:

```
stack-size is the amount of stack space  
file-name is the name of a C source file
```

The compiler, like any C program, requires two fixed size areas of memory called the stack and heap. The stack is a fixed size area of memory reserved for the compilers internal variables. The heap is a fixed size area of memory that is used to store a symbol table for the program being compiled. The compiler enters every unique identifier in a program into the symbol table as it compiles.

The stack-size parameter is an optional parameter that allows you to specify the amount of stack reserved for the compilers internal variables. The stack-size is expressed as an integer number of bytes that may optionally be followed by the letter K. The K is a multiplier corresponding to the value 1024. For example, 4K is equivalent to 4096 bytes (4 * 1024).

The amount of stack space required by the compiler varies from one program to the next. A program containing complex expressions will require more stack space. Expressions nested inside several levels of parentheses cause the

compiler to use a lot of stack space. The angle brackets <> must be used if the stack-size is specified. The default value of the stack-size parameter is 4608 bytes. This is enough stack space for most programs. The minimum amount of stack used by the compiler is 4096 bytes.

All remaining memory (not used as stack) is allocated to the heap. The amount of heap required by the compiler also varies from one program to the next. A program with lots of identifiers (constants, variables, and functions) will require more heap. It is the number of identifiers, not the number of lines, that determines how much heap is required to compile a program.

When the short form is used, the compiler always sends the program listing to the crt. At the end of a compile, the compiler displays the amount of stack and heap used. If there is not enough stack or heap to compile a program, the compiler will terminate and display the amounts used. The stack-size parameter may be used to adjust how the memory is partitioned between the stack and heap. If a compile is terminated due to not enough stack, the stack-size parameter should be used to specify more stack. If the termination is due to not enough heap, the stack-size parameter should be used to specify less stack.

When the compiler is executed using the short form, the file-name should not specify an extension. The compiler appends the extension C to the file-name. Even if an extension is specified, the compiler will ignore it and instead use the C extension. The file name may include a drive specifier to cause the compiler to search a specific drive for the source file. The compiler then creates an object file with the same name as the source file except that the extension OBJ is appended. When the file name includes a drive specifier, the object file is placed on the specified drive.

Example:

```
CC <4K> TEST:1 Allocates 4096 bytes of stack and compiles
                the program in file TEST/C on drive 1,
                creating an object file TEST/OBJ on drive 1.
```

2.1.2 Long Form

With the long form, the compiler prompts for the complete name (including extension) of the files to use for the source, listing, and object.

```
CC <stack-size>
SOURCE = file-name
LISTING = file-name
OBJECT = file-name
```

where: file-name is the complete name of a file
 SOURCE is the input file containing the C program
 LISTING is the output file containing the listing
 OBJECT is the output file containing the object code

The file names are used as specified. Device names may also be used. This form provides more versatility. For example, the source and object can be on different disk drives and the listing can be sent to a file or to the line printer.

Example:

```
CC <4000>           Allocates 4000 bytes of stack and compiles
SOURCE = TEST/C:1   the program in TEST/C on drive 1, sending
LISTING = :L        the listing to the line printer
OBJECT = TEST/OBJ:2 object to TEST/OBJ on drive 2.
```

2.1.3 Compiler Listing

The C compiler produces a listing of the program as it compiles. The listing contains the text of the source program with some additional information.

The listing is divided into pages. At the top of each page is a heading. The heading contains the version number of the compiler, and the page number. Each page after the first contains a form feed (control/L or #0C) character. The form feed will cause a page eject on most printers. The number of lines per page may be changed by a compiler option in the source program. See the Reference Manual.

If errors are detected by the C compiler, error messages will appear in the listing. Error message lines have a string of five asterisks ('*****') at the beginning of the line. An up arrow will appear pointing to the approximate location within the line where the error was detected. This will be followed by one or more error codes. It is possible for a single error to generate more than one error code. In most cases, the first error code identifies the cause of the error.

If any errors are detected, a summary of the meanings of the error codes generated is printed at the end of the listing. Also, an error file named C/ERR is created that contains a list of the C source lines that had errors. If an error occurs on a line containing reference to a macro definition (#define), the expansion of the macro is also displayed on the listing to aid in determining the reason for the error.

2.2 Using the Run Utility

The run utility loads and executes a single object file. The run utility contains the object code for the standard C functions from the following libraries: SYSTEM, CLIB, PRINTF, and SCANF. Therefore, it can be used to quickly load and execute a compiled program without manually linking the libraries to the program. If there are unresolved references in the object file, the file named RUNC/OBJ is loaded in an attempt to resolve the references. This allows the run utility to be used even when performing separate compilations. Simply place the object code of the separately compiled functions in the file RUNC/OBJ. The RUNC/OBJ file supplied contains the functions from the following libraries: TRSLIB, RANDOM, and STRINGS.

```
RUNC <stack-size> file-name
```

where: stack-size is the amount of stack space
file-name is the name of the object file

The file-name may or may not specify an extension. If none is specified, then the extension OBJ is appended to the file name. Otherwise, the file name is used as specified. The file name may also include a drive specifier.

C programs use the stack to store global and local variables and return values for function calls. This stack is allocated when the program is executed and the required size is determined by the number and type of variables declared and the number and sequence of function calls. The heap is used to store file descriptors and dynamically allocated variables.

The optional stack-size parameter may be used to adjust how the available memory is allocated to the stack and heap. By default, half the available memory is allocated to the stack and half to the heap. The stack is specified as an integer number of bytes that may optionally be followed by the letter K. As explained for the compiler, the K is a multiplier with a value of 1024. If a program terminates with an OUT OF STACK error message, then the stack-size parameter should be used to specify more stack. If a program terminates with an OUT OF HEAP error message, then the stack-size parameter should be used to specify less stack.

Example:

```
RUNC <15K> TEST:1  Allocates 15360 bytes for the stack  
                  and executes TEST/OBJ on drive 1.
```

The RUNC command also allows redirection of the standard files stdin and stdout. Following the file name, the symbols < and > may be used to redirect stdin and stdout respectively. Either a device or file name may be used to

redirect stdin and/or stdout.

Example:

```
RUNC TEST <INPUT/DAT >OUTPUT/DAT
```

2.3 Using the Linking Loader

The linking loader on the disk labeled System3 (copied from "Disk 2 of 3") provides facilities for configuring C programs. It provides the ability to link together separately compiled functions. Programs may be linked and stored as command files on disk and then later executed from the operating system as commands. These command files behave in the same way as the utilities supplied with the operating system.

The linking loader is executed by simply typing LINKLOAD. The linking loader displays a menu of commands and waits for a command to be selected.

```
L=Load, R=Run, E=Exit, I=Init, S=Symbols, B=Build CMD  
F=Find in library  
>>
```

All commands require only the single letter, although longer names will also be accepted. To execute a command, simply type its first letter followed by the enter key. If more information is required, additional prompts will be supplied. You may type H and press the enter key to redisplay the menu of commands at any time.

2.3.1 Load Command

The load command is used to load object files into memory. To load an object file, type "L" and press the "enter" key. The load command will prompt with "File(s) =". Type the name of one or more files in standard notation, including extension. Multiple file names must be separated by commas. You may optionally follow the "L" command with the file names to avoid the prompt.

The "L" command opens the object file(s) and loads the object code into memory. Each time a function is loaded, its name is displayed on the screen. The names of global variables are also displayed. This allows you to monitor the load process as the object file is loaded into memory. The names of variables or functions declared as "static" are not displayed.

The object code for each C function is compiled into a separate entity. These are then linked together when they are loaded. This allows functions to be compiled separately and then linked. Thus, a program may be compiled a

piece at a time, and when changes are made, only the parts affected by the change need to be recompiled. This also allows the creation of libraries of utilities. These utilities can be loaded with any program that needs them, but need be compiled only once. The supplied libraries of standard C functions are an example of such utilities.

In general, the functions within an object file may be split into separate object files (See the ADP Manual for details). However, it should be noted that if an object file contains a "static" function, the individual functions within that file may not be separated.

2.3.2 Find Command

The Find command "F" is identical to the "L" command except that only referenced (called) functions within an object file are loaded. This command should be used when linking the supplied function libraries to your programs. The "F" command loads only those functions which your program uses.

With the "L" command, the order in which the functions are loaded is not important since all functions are loaded. However, the order is important with the "F" command. When linking your programs with the supplied function libraries, you should load your program first. Next load any of the libraries needed, excluding CLIB and SYSTEM. The last libraries loaded should be CLIB, followed by SYSTEM. This order insures that a function is referenced before the object file in which it resides is loaded.

One of the files provided on the disk labeled C3 (Disk 3 of 3) is CSUPPORT/BIN. This file contains all the C runtime libraries. It was created by appending all the libraries into a single file (with CLIB and SYSTEM appended last). The file was then processed by HEXTOBIN to translate the libraries to binary format. This file may be used if desired rather than loading the individual libraries separately.

When writing a program consisting of more than one function, or creating your own library of functions, you should arrange each function so that all references to it precede its definition. For example, if function A calls function B, make sure that the definition of function A precedes that of function B. Otherwise, you will have to load the program or library more than once to resolve all references.

2.3.3 Symbols Command

The linking loader stores the name and address of each global variable and function as the object file is loaded. Also stored are the names of functions that have been called (referenced) by another function, but have not yet been loaded into memory. This symbol table can be displayed with the "S" command.

The symbols command displays all currently defined or referenced symbols. The display is paused if the screen fills and the prompt "* MORE *" appears at the bottom of the screen. Press the enter key to view the remaining symbols.

One function name is displayed per line. After the function name is a character that describes the use of that function. A "D" indicates that the name is defined. This means that the function has been loaded into memory. An "R" indicates that the function has been referenced but not yet defined. This means that a function that has already been loaded makes a call to this function. All functions that are called must be loaded before the program can be executed. Global variables will always be displayed with a "D" tag.

The last item on the line is the address of the symbol. If the symbol is defined ("D"), then this is the address in memory where the function or global variable is located. If the symbol has not been defined ("R"), then this is the address of the last place it was used (called).

2.3.4 Run Command

After all the object files of a program have been loaded, it can be executed with the "R" command. The linking loader prompts for the amount of stack space required by the program. As in the RUNC utility, the default is to allocate half of the unused memory to the stack, and the other half to the heap. If these space allocations are sufficient, then simply press the enter key. Otherwise enter a value. As with the RUNC utility, the letter K may be used as a multiplier of 1024 when specifying the stack.

I/O redirection can also be used with the Run command of LINKLOAD. The redirection must be specified when the linking loader is first executed.

```
LINKLOAD >:L    maps stdout to the line printer
```

2.3.5 Build Command

Once a program has been loaded, the "B" command can be used to store the program to disk as an executable command file. Like the Run command, the Build command prompts for the stack size. The next prompt asks for a file name. This is the name of the file that will contain the program. You must type in a file name with a CMD extension. For example, PAYROLL/CMD might be the name of the command file. The B command causes the program to be saved to disk in command file format. The linking loader terminates after building the command file.

The program may then be executed by simply typing the name of the command file. I/O redirection may be specified after the command file name. The < symbol redirects stdin while the > symbol redirects stdout.

```
PAYROLL <WAGES/DAT >REPORT/DAT
```

2.3.6 Init Command

The "I" command clears the symbol table and redisplay the command menu. This command may be used if the wrong file is loaded by mistake. It is equivalent to exiting to the operating system and then executing the linking loader again.

2.3.7 Exit Command

The "E" command causes the linking loader to terminate and return to the operating system.

2.3.8 Error Messages

*** CANNOT OPEN FILE

This message is generated when the loader cannot find the file specified with the "L" command. This may be caused by a misspelling or the wrong disk being in the drive.

*** UNRESOLVED REFERENCES

When the "R" command is used to execute a program or the "B" command to generate a command file, the loader checks that all of the functions that are called within the program have been loaded. If there are functions that have been called but have not been loaded, then this message is generated. At this point, you can load the required files and repeat the command. The "S" command may be used to list names of the functions that are not yet defined. These will be followed by an R tag.

*** INVALID OBJECT TAG

This message is displayed when a load is attempted on a file that is not a valid object file. The most frequent cause of this error is an attempt to load the source program instead of the object.

***** ILLEGAL REFERENCE**

This message signifies an inconsistent structure in an object file. It is an indication that the file has been damaged. The best solution is to recompile the offending program.

Chapter 3

Miscellaneous

3.1 Compiler Memory Constraints

The size of source file that can be compiled is limited by the number of identifiers rather than the number of lines in the file. The compiler allows enough space for approximately 200 identifiers (symbols) to exist in a program being compiled. Each identifier used in a program requires an entry in the compilers symbol table. Typedef, variable, and function names are all identifiers in C programs and are entered into the compilers symbol table. The macro names and definitions used in the #define are also entered into the symbol table. Excessive use of #define will limit the size of source file that can be compiled.

3.2 Runtime Memory Usage

The LINKLOAD and RUNC programs load at address 0x3000. The object code for C programs load immediately above the loader. The next segment above the object code contains the stack. The stack grows from high memory to low memory. The remainder of the available memory is used as the heap.

The heap is a section of memory that is used for dynamic storage. Programs that use the function "calloc" will use storage from the heap. Also, the dynamic string library functions use storage from the heap. The heap also contains the buffers used to read and write to files. The C runtime support routines perform blocking on data from files. Each file is allocated a 256 byte buffer from the heap and information is read or written to this buffer before being transferred to disk. This improves performance by decreasing the frequency of disk accesses.

3.3 Accessing Arguments via Pointers

The arguments to a function may be accessed through a pointer. This is useful for functions that accept a variable number of arguments. Printf and scanf are examples of functions that accept a variable number of arguments. Both of these functions only declare one argument, the format string, even though many arguments are typically passed.

When arguments are passed to a function, they are pushed onto the stack. The first argument is pushed, followed by the second, etc. Since the stack grows from high memory to low memory, the first argument is at a higher memory address than the last argument.

Using a pointer corresponding to the address of the first argument, the remaining arguments are accessed by decrementing the pointer. You must know the size of each argument in order to access them in this manner.

3.4 Size and Range of Basic Types

The size, range, and accuracy of the basic data types are defined in the following table.

type	size in bytes	range of values	accuracy
----	-----	-----	-----
char	1	'\0' to '\377'	
short	1	-127 to 127	
int	2	-32768 to 32767	
unsigned	2	0 to 65535	
(pointer)	2	0x0000 to 0xFFFF	
long	4	-2147483648 to 2147483647	
float	4	+/- 1.7e-38 to +/- 1.7e+38	6 digits
double	8	+/- 1.7e-38 to +/- 1.7e+38	16 digits

Note: All mathematical functions are performed in double precision and have an accuracy of approximately 9 digits.

3.5 Generating EOF from the Keyboard

When receiving input from the keyboard, the EOF character is the backquote (`). On the Model 4, backquote is generated by simultaneously pressing the SHIFT and @ keys.

3.6 Linking Assembly Language

Assembly language subroutines can be assembled using any available assembler and linked to a C program.

The TRSLIB CALL\$ function is used to call assembly language subroutines. The first parameter to CALL\$ is the address of the assembly language subroutine. The remaining parameters of CALL\$ allow you to define values for the Z-80 registers. The CALL\$ function transfers control to the assembly language routine at the specified address. The C program regains control when a Z-80 return instruction is executed. The C program should define values for the registers that the subroutine uses as input parameters. All register values are returned to the C program through the argument list of CALL\$.

The LINKLOAD utility cannot load object files created by your assembler. Therefore the assembly language subroutines must be originated to load at specific addresses. The next section, Patches, has the appropriate address at which assembly language subroutines may begin loading. In the C program, the first argument to CALL\$ must then specify the load address when calling a subroutine.

The LINKLOAD utility must be patched to change the address at which it starts loading object files. This reserves space for the assembly language subroutines. The next section, Patches, explains how to patch the LINKLOAD utility so that space is reserved for the assembly language subroutines.

The assembly language subroutines must be loaded prior to executing the linking loader. Use any appropriate mechanism to load the assembled subroutines. If your assembler generates command (CMD) files, the operating system LOAD command will load the subroutines. Once all the subroutines are loaded, execute the patched version of the linking loader and load in the C object files. You may then Run the program or Build a command file.

3.7 Patches

Normally when a program built with the linking loader terminates, the last instruction address along with the amount of stack and heap used is displayed. After a program has been completely debugged this information is no longer needed. The following patch will eliminate these messages. The patch should be applied to a copy of the linking loader. After the patch is applied, any command file built with the patched copy of the linking loader will not print the stack and heap message.

1. Make a copy of the linking loader.

```
TRSDOS Ready
COPY LINKLOAD/CMD TO LASTLINK/CMD
```

2. Apply the patch using the TRSDOS PATCH command.

```
TRSDOS Ready
PATCH LASTLINK/CMD (X'3915'=01)
```

To link assembly language subroutines, the LINKLOAD utility must be patched to change the address at which object files are loaded. The assembly language subroutines can begin loading at address 0x80DF. The linking loader must then be patched to start loading above this address, providing the space needed for the subroutines. For example, if the subroutines require 256 bytes of space, the linking loader can be patched to start loading at address 0x81DF. If they require 512 bytes of space, the starting load address should be 0x82DF. There is no problem with providing more space than needed for the assembly language subroutines. However, if too little space is provided, the loader will load object files over the assembly language subroutines.

The following patches change the upper byte of the address at which the linking loader begins loading object files. The value of this byte may be changed to allow however much space is required by the assembly language subroutines.

1. Make a copy of the linking loader.

```
TRSDOS Ready
COPY LINKLOAD/CMD TO LINKASM/CMD
```

2. Apply the patches using the TRSDOS PATCH command.
The following patches provide 512 bytes of space for assembly language subroutines between the addresses 0x80DF and 0x82DF. Notice that two separate patches must be applied. The first patch is always the same while the second varies with the

amount of space required for the assembly language.

TRSDOS Ready

PATCH LINKASM/CMD(X'77F4'=21 00 00:X'7749'=C9)

PATCH LINKASM/CMD(X'77E3'=DF 82:X'7414'=DF 82)

Chapter 4

Function Libraries

TRS-80 C is supported by 7 libraries of functions: SYSTEM, CLIB, PRINTF, SCANF, TRSLIB, RANDOM, and STRINGS. The supplied file named CSUPPORT contains all 7 of these libraries. All C programs implicitly use some of the functions in both SYSTEM/OBJ and CLIB/OBJ. The other libraries are not required unless a function in them is explicitly called by the program.

The first 4 libraries contain the standard C functions. SYSTEM/OBJ contains the runtime system interface routines and the low level C library functions. CLIB/OBJ contains most of the standard C library functions. PRINTF/OBJ contains the standard C functions, printf, sprintf, and fprintf. SCANF/OBJ contains the standard C functions, scanf, sscanf, and fscanf.

The next 3 libraries contain non-standard C functions. TRSLIB/OBJ contains functions which provide access to specific Model 4 or TRSDOS 6 features. RANDOM/OBJ contains random access file functions. STRINGS/OBJ contains dynamic string functions.

C programs may be executed with either the RUNC or the LINKLOAD utility. The RUNC utility contains all the functions from the 4 standard C libraries. When executing a program with RUNC, any library function used by the program is automatically linked. The LINKLOAD utility does not contain any of the functions in the 7 libraries. When executing a program with LINKLOAD, any library function used by the program must be linked by loading the library file which contains the function. LINKLOAD and the libraries are on the disk previously configured and labeled as System3.

Each set of library functions is described in the following pages. Any function that is not declared to return a value of type int must be externally declared by the program that uses it. For example, "void SVC();" should be used to declare the TRSLIB function SVC. The special type "void" is used to declare functions that do not return values. Also note that some of the library functions have arguments declared to be of type char. Normally this is not possible in C because there is no way to pass an argument of type char. This is due to the conversion rules for C expressions. Values of type char are automatically converted to type int before being passed. The compiler option /*\$NO CONVERT*/ prior to a function call prevents the automatic type conversion. Therefore, when using a library function that has an argument declared as type char (pointer to char is OK), the /*\$NO CONVERT*/ compiler option must precede the call to the function. After the call, the /*\$CONVERT*/ compiler option may be used to turn automatic type conversion on again.

4.1 SYSTEM Library

The SYSTEM library contains the functions that interface to the C runtime. It includes the functions (getc, putc, cfree, calloc, cfree, ftoa, atof, atan, log, exp, sqrt, sqr, sin, cos, and abs), which are described in the Reference Manual. It also includes several functions which are equivalent to functions in TRSLIB, (_svc, _call, _hperr, _ioerr, _setacnm, _cmdline). In addition, it contains the buffered I/O functions (read and write).

4.1.1 read

```
read(fd, buffer, n)
int  fd, n;
char *buffer;
```

The read function provides buffered input. The fd argument is the file descriptor number that specifies the file from which the input is received. For example, stdin->fd is the file descriptor number for stdin. The buffer argument is a pointer to where the input data is stored. The n argument is the number of characters that are input from the file and stored in the buffer. The read function returns the number of characters that were actually input from the file. A return value of 0 indicates the end of file. A return value of -1 indicates that an error occurred during the read operation.

4.1.2 write

```
write(fd, buffer, n)
int  fd, n;
char *buffer;
```

The write function provides buffered output. The fd argument is the file descriptor number that specifies the file to which the output is sent. For example, stdout->fd is the file descriptor number for stdout. The buffer argument is a pointer to where the output data is stored. The n argument is the number of characters that are output from the buffer to the file. The write function returns the number of characters that were actually output to the file. A return value of -1 indicates that an error occurred during the write operation.

4.2 CLIB, PRINTF, & SCANF Libraries

The CLIB, PRINTF, and SCANF libraries contain the functions for which C source code is provided. The PRINTF library contains the standard C functions (printf, fprintf, and sprintf). The SCANF library contains the standard C functions (scanf, fscanf, and sscanf). The CLIB library contains the remainder of the C functions, including a function called `_initio` that is called at the start of execution for all C programs. The functions in these three libraries (excluding `_initio` and the Unix system interface functions: `open`, `creat`, `close`, `unlink`, and `lseek`) are described in the Reference Manual.

4.2.1 `_initio`

```
_initio(stdfiles)
int    stdfiles
```

The `_initio` function is called at the start of execution of every C program to allocate the file table, open (`stdin`, `stdout`, & `stderr`), and perform I/O redirection. The `stdfiles` argument is used to control whether or not the standard files are opened. If the main function of the program is named "main", then the `stdfiles` argument is 1, the standard files are opened, and the main function is called. If the main function of the program is named "`_main`", then the `stdfiles` argument is 0, the standard files are not opened, and the main function is called. No redirection is performed unless the standard files are opened.

4.2.2 `open`

```
fd = open(name, rwmode);

int  fd;          /* file descriptor      */
char *name;       /* name of the file to open */
int  rwmode;      /* access mode          */
```

Description:

The open function uses the first argument as the name of a file and attempts to open the file. The rmode argument specifies how the file will be accessed. A value of 0 specifies read only, 1 specifies write only, and 2 specifies read/write. It is an error to try to open a file that does not exist.

Returns:

```
fd = file descriptor (file number) if successful
fd = -1 if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    int    fd;
    int    c;
    FILE   *fp;
    fd = open("test/dat",0);
    if (fd != -1) {
        fp = _iob[fd];
        while ((c = getc(fp)) != EOF) putchar(c);
    }
}
```

4.2.3 creat

```
fd = creat(name, pmode);

int  fd;          /* file descriptor      */
char *name;       /* name of the file to open */
int  pmode;       /* protection mode         */
```

Description:

The creat function uses the first argument as the name of a file and creates the file. If the file already exists, then it is truncated to 0 length. It is not an error to creat a file that already exists. The pmode argument specifies the protection mode for the file. This has no effect under TRSDOS 6.

Returns:

```
fd = file descriptor (file number) if successful
fd = -1 if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    int    fd;
    char   c;
    FILE   *fp;
    fd = creat("new/dat",0);
    if (fd != -1) {
        fp = _iob[fd];
        for (c='a'; c<='z'; c++) putc(c,fp);
    }
}
```

4.2.4 lseek

```
pos = lseek(fd, offset, origin);

long pos;          /* current position in file */
int  fd;           /* file descriptor          */
long offset;      /* desired relative position */
int  origin;      /* location to position from */
```

Description:

The lseek function is used to randomly position to any character within a file. The file must have been opened by the open function using an rmode of 2 (read/write access) in order to use the lseek function. The offset argument specifies how many characters (relative to some starting position) that the files pointer is moved. The origin argument specifies the starting position. The origin may be any one of three values. A value of 0 indicates the positioning occurs relative to the beginning of the file, 1 indicates relative to the current position, and 2 indicates relative to the end of the file. You must be sure that the offset value passed to lseek is a long integer. When using integer constants, append the L suffix to the constant.

Returns:

```
pos = current position of file if successful
pos = -1 if unsuccessful
```

Example:

```
main()          /* append example */
{
    int    fd;
    fd = open("old/dat",2);
    lseek(fd, 0L, 2);
    write(fd, "This is added to the end of the file", 36);
}
```

4.2.5 close

```
status = close(fd);

int    status;    /* return status          */
int    fd;        /* file descriptor      */
```

Description:

The close function closes the file associated with the file descriptor fd. This frees the file descriptor for use with another file.

Returns:

```
status = 0 if successful
status = -1 if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    int    fd;
    char    c;
    fd = creat("new/dat",0);
    if (close(fd) != -1) {
        fd = open("new/dat",2);
        write(fd, "abcdefghijklmnopqrstuvwxy", 26);
        close(fd);
    }
}
```

4.2.6 unlink

```
status = unlink(name);

int  status    /* return status    */
char *name;    /* name of file to delete */
```

Description:

The unlink function removes the file specified by name from the directory. The file should be closed before calling unlink.

Returns:

```
status = 0 if successful
status = -1 if unsuccessful
```

Example:

```
main()
{
    if (unlink("new/dat") != -1)
        puts("new/dat deleted");
}
```

4.3 TRSLIB Library

The TRSLIB functions are provided for interfacing to some of the specific features of the Model 4 and the TRSDOS 6 operating system. These functions are shown in the form of function definitions and should be declared or called accordingly.

4.3.1 SVC

```
void SVC(a, status, bc, de, hl, ix, iy)
char *a, *status;
int *bc, *de, *hl, *ix, *iy;
```

(same as `_svc` in SYSTEM Library)

SVC is used to make TRSDOS 6 supervisor calls. Supervisor calls provide the mechanism for executing various TRSDOS 6 operating system routines. See the Technical Reference Manual (Cat. No. 26-2110) for an explanation of the available supervisor calls.

The arguments passed to SVC will be loaded into the Z-80 registers. The arguments will also return the values of the Z-80 registers when the SVC routine terminates. The A register is used to specify an SVC number which determines which operating system routine is executed. Each operating system routine has specifications for which Z-80 registers are used to pass information.

```
main()
{
    void SVC();
    char a, status;
    int bc, de, hl, ix, iy;
    puts("This program displays directories");
    do {
        printf("Enter the drive number: ");
        scanf("%d", &bc);
        a = 34; /* @DODIR from 26-2110 */
        if (bc > -1 && bc < 8) SVC(&a,&status,&bc,&de,&hl,&ix,&iy);
    }
    while (bc > -1 && bc < 8);
}
```

4.3.2 TIME

```
void TIME(s)
char s[8];
```

TIME assigns the string *s* the time of the system clock in the form hh:mm:ss. This function does not terminate the string with a '\0'.

4.3.3 DATE

```
void date(s)
char s[8];
```

DATE assigns the string *s* the date of the system clock in the form mm/dd/yy. This function does not terminate the string with a '\0'.

4.3.4 SOUND

```
void SOUND(tone, duration)
int tone, duration;
```

SOUND is used to generate sound using specified tone and duration codes. The TONE argument should be passed as a number between 0 and 7 with 0 being the highest tone and 7 being the lowest. The DURATION argument should be passed as a number between 0 and 31 with 0 being the shortest and 31 being the longest.

4.3.5 CMDLINE

```
void CMDLINE(location, origin)
char *location, *origin;
```

(same as `_cmdline` in SYSTEM Library)

The CMDLINE function returns pointers to the command line stored by the operating system. Each time a command is executed from the TRSDOS Ready prompt, all characters typed are stored in a buffer within the operating system. For example, when RUNC DATABASE FILE1 <enter> is typed, the operating system stores RUNC DATABASE FILE1 in the buffer. The origin argument returns the address of the beginning of the command line buffer. The location argument returns the address of the character in the buffer that begins with the first non-blank character following the command name.

Using the above command line as an example:

```
origin   points to RUNC DATABASE FILE1
location points to DATABASE FILE1
```

4.3.6 USER

```
void USER(address, data)
char *address;
int  *data;
```

This function interfaces to assembly language routines resident in memory. "Address" points to the location of the first instruction of the routine.

Information is passed to the assembly language routine through the "data" argument. When the assembly language routine is entered, the HL register pair contains two bytes loaded from the location pointed to by "data". When the routine exits, the two byte value in the HL register pair is stored at the location pointed to by "data".

The assembly language routine is entered with a standard Z80 call instruction and should be exited via a return. All Z80 registers are available for use in the assembly language subroutine. The Z80 stack may also be used as long as it is restored to its entry condition before the routine is exited.

4.3.7 CALL\$

```
void CALL$(address, a, status, bc, de, hl, ix, iy)
char *address, *a, *status;
int  *bc, *de, *hl, *ix, *iy;
```

(same as `_call` in SYSTEM Library)

This function can be used in a similar manner to USER to call assembly language subroutines. The difference is that CALL\$ permits you to set up all of the Z80 registers from C. The values passed (except status) will be in the registers when the subroutine is entered. When the subroutine returns, the current contents of all registers are stored at the locations pointed to by the arguments. Status is the Z80 flag register.

4.3.8 \$MEMORY

```
void $MEMORY(stack, heap)
int  *stack, *heap;
```

This function allows a program to determine the amount of memory currently available. The argument "stack" returns the current number of bytes remaining in the stack and the argument "heap" returns the number of bytes remaining in the heap.

4.3.9 HP\$ERROR

```
void HP$ERROR(newstate, oldstate)
char  newstate, *oldstate;

(must use compiler option /$NO CONVERT*/)
(same as _hperr in SYSTEM Library)
```

This function sets the state of the heap error recovery flag within the C runtime system. By default, this flag is set to 0 (false). When the flag is set to a binary 1 (true), a fatal runtime error occurs if there is an attempt to allocate more space from the heap than is available. The fatal error causes the program to terminate. The "newstate" argument contains the value to which the flag is set. The value of the flag prior to the call is stored at the location pointed to by "oldstate".

4.3.10 PEEK

```
char PEEK(address)
char  *address;
```

This function returns the contents of any memory location. It may be used to examine memory or memory mapped input devices. The "address" argument points to the memory location. The contents of this location (one byte) is returned.

4.3.11 POKE

```
void POKE(address, value)
char *address, value;
```

(must use compiler option /*\$NO CONVERT*/)

POKE is used to alter the contents of any location in memory. It may also be used to write to memory mapped output devices. The "address" argument points to the memory location. The "value" argument contains the byte which is stored at that location.

4.3.12 INP

```
char INP(port)
char port;
```

(must use compiler option /*\$NO CONVERT*/)

This function performs input from a Z80 IO port. The "port" argument contains the Z80 port number. A one byte value is read from the Z80 port and returned.

4.3.13 OUT

```
void OUT(port, value)
char port, value;
```

(must use compiler option /*\$NO CONVERT*/)

This function performs output to a Z80 port. It may be used in conjunction with the function INP to communicate with devices interfaced as input or output ports. The "port" argument contains the Z80 port number and the "value" argument is written to that port.

4.3.14 WRITECH

```
void WRITECH(ch)
char ch;
```

(must use compiler option /*\$NO CONVERT*/)

This function writes a single character to the terminal. The argument "ch" is output to the crt. This function may be used to save space by avoiding the creation of a file descriptor.

4.3.15 WRITESTRING

```
void WRITESTRING(s, first, last)
char s[];
int first, last;
```

This function writes a portion of a string of characters to the crt. "First" is the index of the first character to be written. The index for the string starts at 1. "Last" is the index of the last character to be written. The total number of characters displayed is (last - first + 1). If last is less than first, no characters are written. Like WRITECH, this function saves space by avoiding the creation of a file descriptor.

4.3.16 INKEY

```
void INKEY(ch, ready)
char *ch, *ready;
```

This function attempts to obtain a character from the keyboard. If a key is pressed at the time the call is made, the character generated by the key is stored at the location pointed to by "ch" and a binary 1 is stored at the location pointed to by "ready". If no key is pressed at the time of the call, the space character ' ' is stored at the location pointed to by "ch" and a binary 0 is stored at the location pointed to by "ready".

4.3.17 GETKEY

```
char GETKEY()
```

This function waits for and returns the next character from the keyboard.

4.3.18 FILE\$STATUS

```
char FILE$STATUS(file)
char *file;
```

This function returns the status of a file. The "file" argument is a pointer to the actual file descriptor. For example, "stdin->file" rather than simply "stdin" must be passed. The function returns the operating system error code for the latest IO (input or output) operation. If no errors have occurred, then zero is returned.

4.3.19 IO\$ERROR

```
void IO$ERROR(newstate, oldstate)
char newstate, *oldstate;

(same as _ioerr in SYSTEM Library)
(must use compiler option /*$NO CONVERT*/)
```

This function sets the state of the IO error recovery flag within the C runtime system. By default, this flag is set to binary 0 (false). If the error recovery flag is set to a binary 1 (true), then any subsequent error during an IO operation will cause a fatal runtime error to terminate the program. The "newstate" argument contains the value to which the IO error recovery flag is set. The value of the flag prior to the call is stored at the location pointed to by "oldstate".

4.3.20 DELFILE

```
void DELFILE(name, status)
char *name,
int *status;
```

This function deletes a file from any disk in the system. The "name" argument points to the TRSDOS name of the file, including (optional) drive specification. The name must be terminated by a carriage return ('\r'). The operating system error code is stored at the location pointed to by "status". The error code is 0 if the delete operation is successful.

4.3.21 RENAME

```
void RENAME(oldname, newname, status)
char *oldname, *newname;
int *status;
```

RENAME changes the name of a TRSDOS file. The "oldname" argument points to the old name of the file. The "newname" argument points to the new name of the file. The names should be valid TRSDOS file names terminated by a carriage return ('\r'). The operating system error code is stored in the location pointed to by "status". An error code of 0 indicates that the rename operation was successful.

4.3.22 SET\$ACNM

```
void SET$ACNM(file, name, length, id)
char *file, *name, *id;
int length;
```

(same as `_setacnm` in SYSTEM Library)

SET\$ACNM is used to associate the name of the physical file or device to a file descriptor. The argument file is a pointer to a file descriptor (eg. `stdin->file`). The name argument is a pointer to a string containing the name of the disk file. The argument length is an integer that specifies the length of the file name. The argument id is a pointer to an 8 character string that specifies the identifier that is displayed if a subsequent I/O error occurs with this particular file. The first character of the identifier must be uppercase.

4.3.23 SETACNM

```
void SETACNM(file, name)
char *file;
STRING *name;
```

The library function SETACNM serves the same purpose as SET\$ACNM but is simpler to use. The file argument is a pointer to a file descriptor (eg. `stdout->file`). The name argument is a pointer to the file or device name. Notice that name is a pointer to a dynamic string as defined in `stdio`. The SETACNM function frees the dynamic string before exiting to recover the space.

4.3.24 CLEARGRAPHICS

```
void CLEARGRAPHICS()
```

This function clears the screen with blanks.

4.3.25 CLEARSCREEN

```
void CLEARSCREEN()
```

This function does the same thing as CLEARGRAPHICS.

4.3.26 GOTOXY

```
void GOTOXY(x, y)
int  x, y;
```

This function positions the cursor on the screen at the specified location. The value of "x" should be in the range of 0 to 79 and the value of "y" should be in the range of 0 to 23. The top left corner of the screen corresponds to x = 0 and y = 0.

4.3.27 NOBLANK

```
void NOBLANK(redisplay)
char  redisplay;
```

(must use compiler option /*\$NO CONVERT*/)

When the Model 4 video screen receives a carriage return ('\r'), the next line after the line containing the cursor is erased. The NOBLANK function is used in conjunction with input files which are connected to the keyboard to prevent the next line from being erased when the <enter> key is pressed. The NOBLANK function must be called with the argument "redisplay" set to binary 1 (true) to prevent the <enter> key from erasing the next line. NOBLANK must be called before the input file is opened (fopen) in order to have any effect. Therefore, the "stdin" file cannot be used unless _main is used to prevent it from being automatically opened.

When a program is executed from a JCL file, an input file connected to the keyboard receives input from the JCL file instead. To prevent this from occurring, NOBLANK('\l') may be executed prior to opening the input file.

4.3.28 READCURSOR

```
void READCURSOR(x, y)
int  *x, *y
```

This function is used to obtain the current position of the cursor on the screen. The column position of the cursor is stored at the location pointed to by "x". The column value will always be in the range of 0 to 79. The row position of the cursor is stored at the location pointed to by "y". The row value will always be in the range of 0 to 23. The position 0,0 corresponds to the upper left hand corner of the screen.

4.3.29 RSETPOINT

```
void RSETPOINT(x, y)
int  x, y;
```

This function clears (turns off) a graphics point on the screen. The position of the point is specified by the "x" and "y" parameters. The "x" argument should be in the range of 0 to 159 and the "y" argument should be in the range of 0 to 71. The position 0,0 corresponds to the upper left hand corner of the screen.

4.3.30 SETPOINT

```
void SETPOINT(x, y)
int  x, y;
```

This function sets (turns on) a graphics point on the screen. The position of the point is specified with the "x" and "y" arguments. The "x" argument should be in the range of 0 to 159 and the "y" argument should be in the range of 0 to 71. The position 0,0 corresponds to the upper left hand corner of the screen.

4.3.31 TESTPOINT

```
char TESTPOINT(x, y)
int  x, y;
```

This function tests the state of a graphics point on the screen. The position of the point is specified with the "x" and "y" arguments. The "x" argument should be in the range of 0 to 159 and the "y" argument should be in the range of 0 to 71. The value returned for the function is a binary 1 (true) if the point is set (turned on) and 0 (false) if the point is cleared (turned

off).

4.3.32 EXTMEM

```
void EXTMEM(operation, bank, localaddress, extendaddress,
            blocksize, status)
char  operation, *localaddress, *extendaddress;
int   bank, blocksize, *status;
```

(must use compiler option /*\$NO CONVERT*/)

The Model 4 can contain up to 128k bytes of memory. This function allows a C program to use the top 64k of memory to store data under program control. For this function to work, at least one bank of 32k must be free (not used by memdisk or some other program). The argument "bank" is used to specify the bank number in extended memory. The two upper banks in a 128k machine are banks 1 and 2.

The "operation" argument tells EXTMEM which operation to perform. The EXTMEM function supplies all needed operations including allocating and releasing banks of memory. The operating system error code is stored at the location pointed to by "status". An error code of 0 indicates that the operation completed successfully. See the TRSDOS 6 Technical Manual for details.

```
operation = '\0';
```

The "bank" of memory is released.

```
operation = '\1';
```

The "bank" is tested to determine its current state. A status of 1 indicates the bank is busy (in use) and a status of 0 indicates the bank is available.

```
operation = '\2';
```

Reserves the selected "bank" of memory and makes it available for use by EXTMEM. The selected bank is marked as being in use.

```
operation = '\3';
```

Copys a block of data from extended memory to local memory. "Extendaddress" points to the block in extended memory. The addresses in extended memory range from 0x8000 to 0xFFFF. "Localaddress" points to the block in local memory. "Blocksize" is the size of the block in bytes. The size of a data structure can be obtained using sizeof.

```
operation = '\4';
```

Copys a block of data from local memory to extended memory. The arguments are the same as for "operation" = '\3'.

The following sample program illustrates use of EXTMEM. An array is stored and retrieved from extended memory.

```
main()
{
    void EXTMEM();
    float r[120], r2[120];
    char  release=0, test=1, reserve=2, retrieve=3, store=4;
    int   i, *status, size=sizeof(float)*120;
    /* turn off automatic type conversion on parameters */
    /*$NO CONVERT*/
    /* allocate bank 1 of extended memory */
    EXTMEM(reserve,1,r,0x8000,size,&status);
    if (status != 0) printf("unable to allocate bank 1\n");
    else {
        for(i=0; i<120; i++) r[i]=i;
        /*store data in extended memory*/
        EXTMEM(store,1,r,0x8000,size,&status);
        if (status != 0) {
            printf("can't store data in memory\n");
            exit();
        }

        /*retrieve the data*/
        EXTMEM(retrieve,1,r2,0x8000,size,&status);
        if (status != 0) {
            printf("can't get data from extended memory\n");
            exit();
        }

        EXTMEM(release,1,r,0x8000,size,&status);
        for(i=0; i<120; i++) printf("%f\n", r2[i]);
    }
    printf("test completed\n");
}
```

4.4 RANDOM Library

Random access files are files in which records can be both read and written in any order. All the records in a random file are the same length. The length can be anywhere between 1 and 256 bytes. All data in a random file record is stored in binary format.

The following C functions are provided to allow random access to the records in a file. There are four random file functions. The OPENRAND function opens a random file, READRAND reads a record from the file, WRITERAND writes a record to the file, and CLOSERAND closes the file.

Each of the functions requires an argument that is a pointer to a 32 byte buffer used as a file descriptor. The OPENRAND function creates the file descriptor and stores it in the buffer. A pointer to this buffer must then be passed to each of the other functions.

4.4.1 OPENRAND

```
void OPENRAND(f, recordlen, name, status)
char *f;
STRING *name;
int recordlen, *status;
```

- f - A pointer to a 32 byte block of memory where OPENRAND stores the file descriptor.
- recordlen - The length in bytes of a record. The size of a record may be determined using sizeof. For example, recordlen = sizeof(int) if a record contains a single integer. The value of recordlen must be between 1 and 256.
- name - A pointer to the dynamic string that contains the name of the random file. For example, name = stods("DATABASE/DAT").
- status - A pointer to an integer that will contain the error status of the open operation. An error status of 0 indicates that the open was successful. Otherwise there was an error in attempting to open the file.

4.4.2 READRAND

```
void READRAND(f, recordnum, data, status)
char  *f;
int   recordnum, *status;
char  *data;
```

- f - A pointer to the 32 byte file descriptor.
recordnum - The random file record number (0 to 32767).
data - A pointer to a block of memory large enough to hold one record. READRAND reads the record specified by recordnum and stores it at this memory location.
status - A pointer to an integer that will contain the error status of the read operation. A returned status of 0 indicates that the read was successful. Otherwise, there was an error in attempting to read from the file.

4.4.3 WRITERAND

```
void WRITERAND(f, recordnum, data, status)
char  *f;
int   recordnum, *status;
char  *data;
```

- f - A pointer to the 32 byte file descriptor.
recordnum - The random file record number (0 to 32767).
data - A pointer to a block of memory containing the data to write to the random file. WRITERAND writes the data to the record specified by recordnum.
status - A pointer to an integer that will contain the error status of the write operation. A returned status of 0 indicates that the write was successful. Otherwise, there was an error in attempting to write to the file.

4.4.4 CLOSERAND

```
void CLOSERAND(f)
char *f;
```

f - A pointer to the 32 byte file descriptor.

4.4.5 Notes and Error Codes

- (1) All blocking is taken care of by the system.
- (2) Detecting the end of file on a random access file is sometimes not exact. The status parameter should be checked after a read to determine if the operating system has detected end of file.
- (3) The function OPENRAND is used to open a file for reading and writing. Opening an empty file and reading is perfectly legal.
- (4) Random file record numbers are defined from 0 to 32767.

Random File Error Codes
Returned By Status Argument

128 - FILE NAME IS NULL OR TOO LONG
129 - RECORD LENGTH TOO LARGE
130 - FILE IS ALREADY OPEN
131 - FILE IS NOT OPEN

Any other returned code is an operating system code.
(See the Model 4 Disk System Owner's Manual)

4.4.6 Example

The following example illustrates the use of the random file routines. The status may be checked after each random file operation to determine if an error occurred. The returned status will be 0 if no error is detected during an operation.

```

#include "stdio"
main()
{
    void OPENRAND(), CLOSERAND(), READRAND(), WRITERAND();
    char file[32];
    STRING *name;
    typedef struct {
        char name[20], address[30];
        int age;
    } record;
    int status;
    int recnum;
    int i;
    record data;
    name = STODS("RANDOM");
    OPENRAND(file, sizeof(record), name, &status);
    checkstatus(status);
    recnum = 0;
    write: printf("\nrecord number %d\n", recnum);
        printf("    enter name : ");
        if (gets(data.name) == NULL) goto read;
        printf("    enter address: ");
        gets(data.address);
        printf("    enter age : ");
        scanf("%d%c", &data.age);
        WRITERAND(file, recnum, &data, &status);
        checkstatus(status);
        recnum++;
        goto write;
    read: printf("There were %d records entered.\n\n", recnum);
        for (i=0; i<recnum; i++) {
            READRAND(file, i, &data, &status);
            checkstatus(status);
            printf("record number %d\n", i);
            printf("    name = %s\n", data.name);
            printf("    address = %s\n", data.address);
            printf("    age = %d\n\n", data.age);
        }
    CLOSERAND(file);
}

checkstatus(status)
int status;
{
    if (status != 0) {
        printf("*** error #%d ***", status);
        exit();
    }
}

```

4.5 STRINGS Library

The following functions are provided for handling dynamic strings. A dynamic string is defined in `stdio` as `STRING`. In these functions, the string arguments and the returned strings are pointers to dynamic strings.

4.5.1 LEN

```
LEN(s)
STRING *s;
```

The `LEN` function returns the length of a string.

4.5.2 LEFT\$

```
STRING *LEFT$(s, position)
STRING *s;
int    position;
```

The `LEFT$` function returns the left portion of the string ending at the specified position within the string.

4.5.3 RIGHT\$

```
STRING *RIGHT$(s, position)
STRING *s;
int    position;
```

The `RIGHT$` function returns the right portion of the string starting at the specified position within the string.

4.5.4 MID\$

```
STRING *MID$(s, position, length)
STRING *s;
int    position, length;
```

The `MID$` function returns the portion of the string starting at the specified position and including the number of characters specified by the argument `length`.

4.5.5 STR\$

```
STRING *STR$(length, ch)
int    length;
char   ch;
```

(must use compiler option /*\$NO CONVERT*/)

The STR\$ function returns a string of the specified length which is filled with the character argument ch.

4.5.6 ENCODEI

```
STRING *ENCODEI(n)
int    n;
```

The ENCODEI function returns a string which is the character representation of the argument n.

4.5.7 ENCODER

```
STRING *ENCODER(r)
float  r;
```

(must use compiler option /*\$NO CONVERT*/)

The ENCODER function returns a string which is the character representation of the argument r (for single precision).

4.5.8 ENCODED

```
STRING *ENCODED(r)
double r;
```

Same as ENCODER, but for double precision reals.

4.5.9 DECODEI

```
DECODEI(s)
STRING *s;
```

The DECODEI function returns an integer number which is the binary representation of the string argument s.

4.5.10 DECODER

```
float DECODER(s)
STRING *s;
```

The DECODER function returns a real number which is the binary representation of the argument s (for single precision).

4.5.11 DECODED

```
double DECODED(s)
STRING *s;
```

Same as DECODER, but for double precision reals.

4.5.12 CHARACTER

```
char CHARACTER(s, position)
STRING *s;
int    position;
```

The CHARACTER function returns the character at the specified position in the string.

4.5.13 CMPSTR

```
char CMPSTR(s1, s2)
STRING *s1, *s2;
```

The CMPSTR function compares the two string arguments and returns a value based on the comparison. The returned value is 0 if $s1 < s2$, 1 if $s1 = s2$, and 2 if $s1 > s2$.

4.5.14 CONC

```
STRING *CONC(s1, s2)
STRING *s1, *s2;
```

The CONC function returns a string which is the result of concatenating the string argument s2 to the end of the string argument s1.

4.5.15 CPYSTR

```
STRING *CPYSTR(s)
STRING *s;
```

The CPYSTR function returns a copy of the string argument s.

4.5.16 DELETE

```
STRING *DELETE(s, position, length)
STRING *s;
int    position, length;
```

The DELETE function returns a string that is comprised of the string argument s after deleting some characters. The number of characters deleted from s is specified by length. The deletion starts at the character specified by position, the first character being position 1.

4.5.17 FIND

```
FIND(subs, s)
STRING *subs, *s;
```

The FIND function returns an integer number corresponding to the starting position of the substring argument subs within the string argument s. The returned value is 0 if subs is not contained within s.

4.5.18 INSERT

```
STRING *INSERT(subs, s, position)
STRING *subs, *s;
int    position;
```

The INSERT function returns a string resulting from inserting subs into s at the specified position within s. The subs string is inserted prior to the character at the specified position within s.

4.5.19 REPLACE

```
STRING *REPLACE(olds, news, s)
STRING *olds, *news, *s;
```

The REPLACE function returns a string resulting after replacing the substring olds with the substring news within string s.

Index

\$memory 33
build command 13
call\$ 32
CC 1, 7
CCB 2, 3
character 48
cleargraphics 38
CLEARSCREEN 38
close 28
closerand 44
cmdline 31
cmpstr 48
CODEGEN 2
compiler listing 9
conc 49
cpystr 49
creat 26
date 31
decoded 48
decodei 48
decoder 48
delete 49
delfile 36
encoded 47
encodei 47
encoder 47
exit command 14
file\$status 36
find 49
find command 12
getkey 36
gotoxy 38
heap 7, 17
hp\$error 33
I/O redirection 10, 13, 13
init command 14
inkey 35
inp 34
insert 50
io\$error 36
left\$ 46
len 46
LINKLOAD 4
load command 11
lseek 27
machine instructions 2
mid\$ 46
noblank 38

open 25
openrand 42
OPTIMIZE 2
out 34
p-code instructions 1
peek 33
poke 34
read 24
readcursor 39
readrand 43
redirection 10, 13, 13
rename 37
replace 50
right\$ 46
rsetpoint 39
run command 13
RUNC 3
set\$acnm 37
setacnm 37
setpoint 39
size of basic types 18
sound 31
stack 7, 17
stack size default 8
stack size minimum 8
str\$ 47
svc 30
symbols command 12
testpoint 39
time 31
unlink 29
user 32
write 24
writech 35
writerand 43
writestring 35
_call 32
_cmdline 31
_hperr 33
_initio 25
_ioerr 36
_setname 37
_svc 30

Table of Contents

Chapter 1 Beginning Concepts	3
The C function and simple output using printf	
Identifiers and reserved words	
Constants and variables	
Data types: char, int, long, float, and double	
Arithmetic operators: +, -, *, /, and %	
Chapter 2 Simple I/O and Expressions	15
Input and output files: stdin, stdout, and stderr	
Format conversions for printf and scanf	
Input using scanf and the address of operator, &	
Defining constants with #define	
Operator precedence and grouping in expressions	
Arithmetic using mixed data types	
Automatic type conversions	
The assignment operators	
The increment and decrement operators	
Chapter 3 Control Statements	29
Relational operators: ==, !=, <, >, <=, >=	
The if and else statements	
Compound statements	
The while statement	
Logical operators: &&, , !	
The for statement	
The do-while statement	
Chapter 4 Functions	41
Calling functions in C	
The return statement	
Function arguments	
Declaring the type of value returned by a function	
Functions that return no value: the type void	
Chapter 5 More I/O and Control Statements	51
Including other files: #include	
Including the standard header file: stdio	
The getc and putc functions, detecting end of file	
The getchar and putchar functions	
The break statement	
The goto statement and statement labels	
The continue statement	
Using ? : in place of if-else	

- Using the * operator to declare pointers
- The hexadecimal format, %x
- Passing arguments by reference
- Declaring arrays
- Referencing array elements, subscripts
- Referencing the whole array
- Arrays of characters, strings
- The string functions, strcpy and strcat
- The string format, %s
- The NULL string terminator
- Initializing character arrays
- The gets and puts functions
- Arrays of more than one dimension
- Using pointers to access array elements
- Using an array of pointers
- The string compare function, strcmp
- Pointers to functions
- Array of pointers to functions
- Initializing arrays in general

Chapter 7 Storage Classes and Scoping

81

- Auto variables, local to a function
- Extern variables, global to all functions
- Definition vs. Declaration of a global variable
- Compiling functions separately
- Static variables, local and global
- Static functions

Chapter 8 Structures and Dynamic Memory

91

- Structure variable declarations
- Referencing members of structure variables
- Naming structure definitions
- Nested Structures
- Pointers to structures and the -> operator
- Arrays of structures
- The sizeof operator
- Dynamic memory allocation, the calloc function
- Structure + dynamic memory = linked list

Chapter 9 File I/O

107

- Opening files, the fopen function
- The getc, putc, fscanf, and fprintf functions
- Type definitions, typedef
- The fgets and fputs functions
- Closing files, the fclose function

Preface

This manual is intended to be an intermediate level tutorial for the C programming language. As such, it is aimed specifically at the programmer who has some experience with a language such as BASIC. This tutorial makes extensive use of program segments and whole programs as examples to guide the reader to a basic understanding of the C language. Some readers may also find it helpful to refer to the C Reference Manual for additional explanations and detail.

This is a full implementation of C as defined in Appendix A of The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie. Extensions to the language are covered in the C Reference manual.

Introduction

The C language has been called a "portable assembler" because programs written in C can be run on a variety of computers with few or no source level changes. The features of the C language give the same level of control over the machine that is found with an assembler. As such, C is an ideal language for use by systems programmers who consider efficiency a major consideration in writing systems level tools. C is also becoming the language of choice of applications programmers. The powerful features of C, together with its portability, make it a very good language for general purpose applications.

The C programming language was designed and written by Dennis Ritchie at Bell Laboratories. It is a descendant of the language B, that was originally written to run under the UNIX operating system, on the DEC PDP-11 computer. Almost all the tools and the UNIX operating system itself are written in the C language. As the usage of the UNIX operating system has spread far and wide, so has the use of C.

The purpose of this tutorial is to guide you to an understanding of the basic characteristics of the C programming language. The objective in choosing the examples provided in this tutorial was primarily to demonstrate the topic of discussion. But they will also give you an idea of the typical uses of the C programming language. While the examples presented in the early chapters do not necessarily meet this second objective, the majority of the remaining examples do represent typical C programs.

The best way to learn a new programming language is to write programs using that language. You are encouraged to enter and execute all the example programs in this tutorial. Then you should experiment with changing the programs to gain experience and confidence in working with C.

Chapter 1

Beginning Concepts

This chapter starts with the basic elements of the C programming language. You'll explore the absolute minimum requirements for a C program. In the process, you will learn about the basic structure of the C function, the way to create output in C, and how to document your intentions within a program using a comment. You'll also be introduced to C's basic data types and arithmetic operators.

The key element in the structure of a C program is the function, so it is important for you to understand a C function thoroughly. A function can be thought of as a black box inside which a task is performed. You really aren't concerned about how the task is accomplished, just that it was performed. For example, let's take a function that calculates the square root of a number. You send that function a number and the function returns the square root of the number. You don't care how it calculated the answer (as long as it is correct), you just want the answer.

All C programs consist of a set of one or more functions. The number and purpose of functions in C programs varies as widely as the weather between the North and South Poles, but every C program will have at least one function called `main`. `main` is a special function, which is considered to be the "driver" of the program. In C programs, execution of the program always begins and ends in the `main` function. In fact, large programs may consist of only a few statements in `main`, which do nothing but direct control from one function to another.

Our first example is a complete but extremely simple C program. When entered with a text editor and compiled with the C compiler, it will have no errors:

Example 1.1

```
main()
{
}
```

This program will also run (execute) successfully, although this may be

difficult to prove. The program doesn't do anything that you can see. The point in presenting this example is to demonstrate the structure of a simple C program.

This program consists of one function called `main`. It, like all C functions, must have a function header (in this case `main()`) and a body enclosed by braces (`{}`). The body in this case is empty.

The function body is also called a block. Anytime you have zero or more C statements enclosed within braces, this is referred to as a block. The C statements that appear inside the block can be very simple or very complex. One thing they all have in common is that they end with a semicolon (`;`). The simplest C statement is the null statement composed only of a semicolon. A block of code containing only the null statement looks like this:

```
main()
{
    ;    /* null statement */
}
```

Notice however, there is no semicolon after the closing braces. The braces tell the compiler to treat the enclosed lines as one unit, but they do not require a semicolon.

Let's now examine a program that does something to let us know when it has run.

Example 1.2

```
main()                /* Example 1.2 */
{
    printf("One small step for C, one giant leap for me.");
}
```

You'll know when this program successfully executes, because the message:

```
One small step for C, one giant leap for me.
```

will be printed on your screen. The line in our program which says:

```
printf("One small.... ");
```

generated the line of print to the screen. This line is an example of a particular C statement, the function call. It consists of the function name followed by a list of values enclosed by parentheses (`()`). The values which are

passed to a function are known as arguments.

In this statement, the function being called is `printf`. It is a special C function that outputs to the screen. You tell `printf` what to output by passing it an argument. `printf`'s argument is a sequence of characters known as a format string:

```
One small step for C, One giant leap for me.
```

You'll recognize it as a C string because it is surrounded by double quote marks (`""`). `printf`'s format string consists of characters to be output and some special format conversion characters. These special characters will be covered later.

More detail about functions and function calls is also presented later. For now, you only need an idea of what functions and function calls are like. Before going on to the next topic however, there is one other item that appears in our sample program which needs to be discussed.

Note that our function call to `printf` ends with a semicolon (`;`). The semicolon denotes the end of this C statement. A statement may be of any length (even written over several source lines), but it cannot span the boundaries of a C block. However, a C string can't span across a line boundary unless a `\` (backslash) character is used just before the end of the line. With this in mind, the previous program could be written:

```
main()
{
    printf(
        "One small step for C, one giant leap for me.");
}
```

Now, let's look at another simple C function:

Example 1.3

```
main()                /* Example 1.3 */
{
    printf("This is the way we wash our face,\n");
    printf("Wash our face,\n");
    printf("Wash our face.\n");
    printf("\n");
    printf("This is the way we wash our face, ");
    printf("so early in the morning.\n");
    /* usually sung to young children as one scrubs
       the face. */
}
```

The output resulting from the preceding program would be:

```
This is the way we wash our face,  
Wash our face,  
Wash our face.
```

```
This is the way we wash our face, so early in the morning.
```

To cause each phrase to begin on a new line, we insert the newline character `\n` (backslash n). The `\n` is an escape sequence representing the ASCII character or characters necessary to generate a new line on your system. There are other escape sequences that will allow you to enter unprintable characters like backspace or tab into the string (See the Reference Manual for details). You might notice that by using the newline character, you can build a single output line using several C statements. For example, the phrase

```
This is the way we wash our face, so early in the morning.
```

is the result of two separate statements:

```
printf("This is the way we wash our face, ");  
printf("so early in the morning.\n");
```

The first call to `printf` writes the first half of the output line. The second call to `printf` finishes the sentence and outputs the newline character(s) for your system.

Also demonstrated in this example is the use of the C comment. Anything appearing between `/*` (slash asterisk) and `*/` (asterisk slash) is ignored by the compiler. Therefore, the line :

```
/* usually sung to young children .... */
```

is a comment, and will have no effect on how the program executes or what it outputs. Comments may be any length, and may be used anywhere in a program. It is highly recommended that you use comments liberally throughout a program. This makes the program easier to understand when you return to modify the program logic.

As you proceed to learn about data types, there are some other important concepts with which you should be familiar:

1. An identifier is a name of a variable or function in a C program. It is a sequence of letters and digits beginning with a letter. The underscore character(_) is counted as a letter. An identifier is case-dependent. That is, upper and lower case letters are different. Here are some examples:

```
/* these are unique id's */
THISID  thatid  MixedUp
newname  NewName  NEWNAME
```

2. Identifiers are allowed to be any length. However, only the first eight characters are significant to the compiler. The following pairs of names are treated as identical identifier names:

```
sameIDas      sameIDasthis
Student_report Student_grade
```

3. Certain words have special meaning to the compiler. These words are known as reserved words. They cannot be used as identifiers. Reserved words may be entered in upper or lower case. So AUTO, Auto, and auto (for example) are all taken to be reserved words. The complete list of reserved words is:

```
auto      double  if          static
break     else     int         struct
case      entry    long        switch
char      extern  register    typedef
continue  float   return      union
default   for     short       unsigned
do        goto    sizeof      while
                          void
```

4. All symbols used in C are referred to as punctuation. The newline, blank (space), and tab characters are referred to as whitespace. When those characters are printed, what shows on your printer or terminal is blank (whitespace). Since some terminal keyboards may not contain all the special symbols used by C for punctuation, there is a set of alternate symbols. Anytime a symbol appears in the C language that is not on the keyboard, substitute the appropriate alternate symbol(s). See the Reference Manual for details.

You should now be comfortable with the concept of a C function. Now, turn your thoughts to what a C function can do. It is the basic building block of C. Each function performs a task by manipulating data. There are many ways to use the word data: data transmissions, data exchange, data retrieved, etc. In this chapter, the word data refers to the values which are used by a program. They appear as constants or variables.

Constants are fixed values, while variables are changeable. The value of a variable can be altered during execution of a program, while the value of a constant will always remain the same. In fact, a constant actually becomes part of the executable code when a program is compiled. Variables, on the other hand, cannot be handled this way because the program would have to be recompiled every time the value of a variable changed.

To circumvent this problem, each variable is assigned a fixed location (address) in memory. Each time the variable is used, the program goes to that location and retrieves the value stored there or stores a new value. The amount of memory space reserved is determined by the variable's declarations.

Let's now turn our discussion to how to declare a variable. Declarations are specified by giving the reserved word for the data type desired, followed by the list of variables to be declared. These declarations must appear first in a block of C code. The data types that can be used in C declarations include: characters, integers, and floating point numbers. The first one you will see is the integer. It is a number which can be negative or positive with no fractional part.

To declare integer variables, you use the reserve word `int`. The following are integer variable declarations:

```
int our_touchdowns, our_field_goals;
int our_score, their_score;
int their_tds, their_fgs;
```

The words `our_touchdowns` and `our_field_goals` are variables specified in what is called a list. In a C declaration you may specify more than one variable of the same type by listing them separated with commas.

Now that you've seen integer variables, let's talk about how integer constants look. They are whole numbers like 5 or 25. These constants can also be negative by writing a minus sign, `-`, in front of the number (`-10`). Integer constants do not have to be declared in any manner before they are used, but they have to be within the machine dependent limits for an integer data type.

Once you've defined the variables in the declarations, you will want to give a value to the variables. This is accomplished by a C statement using the assignment operator (`=`). The value assigned is either a constant, a variable, or the result of an expression.

In C, expressions are a combination of symbols that represent a particular operation (operators) and data values (operands). The operations include arithmetic operations such as add (`+` operator), subtract (`-` operator), multiply (`*` operator), and divide (`/` operator). Any sequence of operators and operands is known as an expression. Some examples of expressions are:

```

b   /* a & b must be declared */
3
a = 1
a + 3
433+b/624*21

```

The first operator that you will see is the assignment operator, =. First, make note that this is not the same as the mathematical equals sign. In this context, it means replacement. A natural tendency is to read a statement like

```
z = 5;
```

as "z is equal to five". In C the correct interpretation is "assign the value of the integer constant five to the variable z". The result of the following statement is to assign the value 24 to the integer variable `our_score`.

```
our_score = 24;
```

Here's a short program example that demonstrates some variable declarations. Also shown in this example is the use of the assignment operator (=). The program first declares some integer variables, assigns some values to those variables, and then prints the values of the variables.

Example 1.4

```

main()           /* Example 1.4 */
{
    /* declare integer variables */
    int our_score;
    int their_score;
    int our_touchdowns, our_field_goals;
    int their_tds, their_fgs;

    /* assign values */
    our_score = 24;
    their_score = 14;
    our_touchdowns = 3;
    their_tds = 2;
    our_field_goals = 1;
    their_fgs = 0;

    /* print the results of the game */
    printf("           Home  Visitor\n");
    printf("Score      %d    %d\n",

```

```

        our_score, their_score);
printf("Touchdowns %d      %d\n",
       our_touchdowns, their_tds);
printf("Field Goals %d      %d\n",
       our_field_goals, their_fgs);
    }

```

Notice in Example 1.4 that the declarations appear before the statements. The statements generate code that is executed when the program is run. The declarations only inform the compiler about the type of variables. They must appear before all executable statements in a block or function.

Before proceeding onto the topic of other variable types, there is something introduced in the above program which needs further explanation. Demonstrated is a call to the formatted print function, `printf`. In the statement,

```
printf("Score      %d      %d\n", our_score, their_score);
```

the percent `d` conversion specification, `(%d)` reserves the place to print an integer variable. The format string is printed exactly as it appears up to the `%d`. At that point, `printf` retrieves the value of the variable named `our_score` and prints it. Then, the rest of the characters of the format string are printed until the second occurrence of `%d` is encountered. The second usage of `%d` in the `printf`'s format string is matched with the variable, `their_score`. Take a look at the output from this statement:

```
Score      24      14
```

Other symbols to use in I/O format strings, like `%d`, will be introduced when the other data types are covered later in this tutorial. For now it is sufficient that you understand that the percent `d` (`%d`) is used in format strings to reserve a place to print integer variables. Notice that this particular call to `printf` has three arguments. Did you also notice that the number of conversion specifications must match the number of additional arguments passed?

What happens if the arguments to `printf` don't match the number of conversion specifications in the format string? If the format string mentions only two values to be printed, but there are three values passed to `printf`, then only two values will be printed. `Printf` only outputs what it is told in the format string. If the format string mentions two values, but only one value is passed, then something unpredictable will be printed (like a large negative number).

Now let's find out more about what the operators can do with our variables and constants. The language C, like almost all other languages, provides a set of symbols called arithmetic operators. The arithmetic operators are used on any numeric data. This includes floating point and character, as well as integer data. The symbols used are

```
+   addition operator
-   subtraction operator
*   multiplication operator
/   division operator
%   modulo operator
```

The only one of these operators which needs any real explanation is the modulo operator(%). Modulo is the "get the remainder of this division" operator. For example, if you divide 23 by 5 you get 4 with a remainder of 3. The modulo operation, 23 % 5, then would give us the result of 3.

A character constant is an ASCII character enclosed by single quotes. It represents a numeric data value in C. The constant 'a' is the representation for the numeric value of the ASCII small letter a, which is 97. For the character 'A', the value is 65, for 'B', the value is 66, etc. A complete list is available in an appendix of the Reference Manual. Non-printable characters may also be represented by an escape sequence. So '\n' is a character constant representing the newline character.

Character declarations are similar to the integer declarations. The following are a few examples:

```
char aa;
char in_char,out_char;
```

Other data type declarations are made in a similar manner. The only difference is the reserve word for the data type. Other reserve words that you can use are long, float, & double. Next you'll take a look at what kind of data is represented in long, float, & double variables.

Floating point data values are numbers that are expressed as fractions, such as 3.4 or 0.00091234. These real numbers can be very large or very small and are sometimes expressed in scientific notation as a power of ten. Examples of real constants:

```
3.4
.91234e-3. (e stands for exponent)
-923.999
0.2345e23
```

To declare floating point variables, you use the reserve word float. Floating point variables allow the maximum range for a numeric value and also allow fractional parts. The accuracy of a floating point value is not always exact. Calculations involving floating point values can result in small errors due to the limited number of digits stored. The small errors can accumulate in some calculations, thus producing an answer that is not correct. The data type, double, can be used to make the numbers more exact. However, the variables declared as double take twice as much memory as floating point numbers.

The last two data types to discuss are long and unsigned. These two data types are both integers. The long data type is used to declare the maximum size for an integer variable. The long integer type typically allows a larger range of values to be stored as integers. The unsigned integer is the same size as the int data type. However, an unsigned integer is capable of storing a value nearly twice as large as a normal integer. The difference is that an unsigned integer can not be negative. Only positive values may be stored in an unsigned variable. The unsigned data type is typically used to store data that is always positive. For example, the number of items in an inventory list can not be negative.

Some important points to remember about declarations are:

1. A declaration of a variable actually reserves a place in memory for its storage.
2. The size of the place reserved is determined by the type specified in the declaration.
3. All variables must be declared before being used in a statement.
4. Declarations must be placed at the beginning of a block.

Now take a look at how a program can use these various types of data.

Example 1.5

```
main()          /* Example 1.5 */
{
    long total, stock;
    unsigned sales;
    float price, commission, cost, income;
    double profit;
    char type;

    sales = 45678; /* items sold */
    stock = 112502; /* items on hand */
    price = 42.50; /* selling price */
    cost = 19.95; /* cost of item */
    type = 'A';

    /* calculate values */
    total = stock - sales;
    income = price * sales;
    profit = (price - cost) * sales;
    commission = profit * 0.03;

    /* print answers */
    printf(
        "total inventory = %ld of type %c\n",total,type);
    printf("On sales of %u items: profit = %f\n",
        sales, profit);
    printf("Commissions at a rate of 0.03 = %f\n",
        commission);
}
```

The program itself is very simple, but note that in the printf statement, some new conversion specifications are used. To print integers you used %d to designate where to place the output value. For long integer variables you use %ld, for unsigned variables you use %u, for floating point variables you use %f, and for character variables you use %c.

Having completed this chapter on beginning concepts, you should understand that all C programs are made up of a set of functions. The C program control always begins and ends in a special function called main and therefore, every program must have a function called main. The basic format of C programs, functions, statements, and comments should be familiar to you by now. You should also have a rudimentary knowledge of C's basic data types and the corresponding constants for each type. With this elementary background, you can now go on to bigger and better programs.

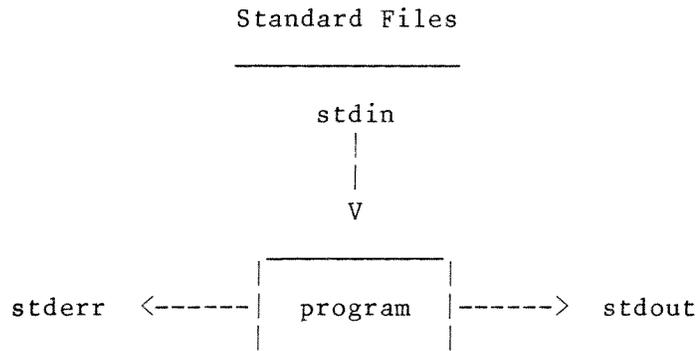
Chapter 2

Simple I/O and Expressions

The C language itself does not provide any input or output statements. You can write your own I/O functions or use the "standard library". This library is implemented by functions that are provided with each C compiler implementation. You've already been introduced to one of those standard functions, `printf`. Let us find out more about C's input and output.

The standard input and output files for a program consist of three files: `stdin`, `stdout`, and `stderr`. The input to a C program is assumed to be a stream of data from `stdin`, the standard input device. `Stdout`, the standard output device is a stream of data output from a C program. The default device for `stdin` is the terminal keyboard, while `stdout` is defaulted to the terminal screen. You can direct these standard files to other devices on your computer system or to a file. However, the method used to redirect I/O is system dependent and will not be covered in this tutorial.

The standard error file is `stderr`. This file is always mapped to the terminal screen. C programmers use this file to report errors in a program. You must however, write messages specifically to this file. How to write to a file, such as `stderr`, will be covered much later in the tutorial. For right now, just be aware that C automatically provides that file.



The standard C functions, `scanf` and `printf`, perform formatted I/O. `scanf` is used for input. It translates character representations of numbers to an internal binary format. `printf` is used for output. It translates binary representations of numbers to an external character format. Earlier, you saw how the format string tells `printf` about the output format. You also discovered that `printf` requires other arguments besides the format string if a conversion specification is used. Well, `scanf` uses a format string and other arguments in a similar manner. It uses the arguments as locations to store the input it processes. The size of each location is determined by the format string conversion specification used.

The conversion specifications used by `scanf` for input are almost exactly the same as those used in the `printf` function. There is one minor exception. The `printf` function uses the conversion characters, `%f`, for a double precision (double) or for a floating point (float) value. `printf` does not need to distinguish between the two because both type arguments look the same when `printf` sees them. You will understand why this is true later when you study conversion of operands in expressions. However, `scanf` must know that it is reading a double precision value so that it stores the correct size data. Therefore, the conversion specification for `scanf` must be `%lf` (long float) for double precision variables. All the conversion characters you have been introduced to are listed in this table as a convenient reference:

Type	Conversion spec.
int	<code>%d</code>
long	<code>%ld</code>
char	<code>%c</code>
float	<code>%f</code>
double	<code>%lf</code> for <code>scanf</code> , <code>%f</code> for <code>printf</code>

Now take a look at a program that calculates the distance traveled using input entered through `scanf`.

Example 2.1

```
main()                                /* Example 2.1 */
{   /* figure distance traveled */
    int mph,time;
    printf("Enter your speed in miles per hour: ");
    scanf("%d",&mph);
    printf("Enter the number of hours: ");
    scanf("%d",&time);
    printf("Miles traveled = %d \n", mph * time);
}
```

It is important to notice that `scanf`'s arguments are preceded by the address-of operator, `&`. Recall that variables are assigned a fixed location in memory. This is where the value of that variable is stored. Since the purpose of `scanf` is to store input values into variables, it must be passed the locations of the variables.

To fully understand why the address-of operator is required, you need to be aware that there are two ways of passing arguments to a C function: call by value and call by reference. When functions are called in C, the variables are passed to the called function by value (call by value). The function only knows the value of the variable and not the location where the variable is stored. This is fine if you do not want to change the value of the variable, but what happens if you do, like for `scanf`? By passing the address of the variable (using the `&` operator) to `scanf`, you can change the value of the variable. This is known as passing the variable by reference. `scanf` does the conversion of the input to its binary representation and then stores the value at the address of the appropriate variable.

In Example 2.1, if by accident you leave off the `&` before the variables `mph` or `time` you will have a problem. `scanf` will be passed the value of `mph`, not its location. `scanf` will then attempt to store the input, using the value of `mph` as a memory address. Consequently, an area of memory, possibly containing the operating system or the executable code of your program, will be destroyed.

Using `scanf` to input from the terminal increases the program's flexibility. Let's take a look at the last example of Chapter 1. It will now take input from the terminal instead of the values being fixed in the program.

Example 2.2

```

main()                /* Example 2.2 */
{
    long total, stock, sales;
    float price, commission, cost, income;
    double profit;
    char type;

    /* input values to be used */
    printf("Please enter the following items ");
    printf("to calculate gross profit,\n net profit,");
    printf(" and sales commision on a item \n");
    printf("Enter the type of item (1 char.) :");
    scanf("%c",&type);
    printf("Enter the number of items sold : ");
    scanf("%ld",&sales); /* items sold */
    printf("Enter the number of items on hand");
    printf(" at the beginning of the month :");
    scanf("%ld",&stock); /* items on hand */
    printf("Enter the current selling price : ");
    scanf("%f",&price); /* selling price */
    printf("Enter the cost of the item : ");
    scanf("%f",&cost); /* cost of item */

    /* calculate values */
    total = stock - sales;
    income = price * sales;
    profit = (price - cost) * sales;
    commission = profit * 0.03;

    /* print answers */
    printf(
        "End of month total inventory = %ld of type %c\n",
        total,type);
    printf("On a gross income of ");
    printf("%6.2f profit = %6.2f\n",
        income, profit);
    printf("Commissions at a rate of 0.03 = %6.2f\n",
        commission);
}

```

Take a close look at the way this program gets its input. Before each `scanf` function call, there is a function call to `printf`. This call outputs a prompting message on the terminal so that you know the program is waiting for input. You also know by these messages, the appropriate data to input. By using `scanf` for input, this program will not have to be recompiled every time

new data is tried. Quite a change from the previous version and certainly easier to maintain!

A new conversion specification, %6.2f, is used in this example. This prints floating point numbers in a field of at least 6 digits in width with 2 digits to the right of the decimal point. If the argument has fewer characters than the field width then the number is padded on the left with blanks.

Another feature of the C language that supports easy changing of program data is the symbolic constant. This feature, #define, allows you to declare a symbolic name for a string of characters. Every place the name occurs, the string of characters is substituted.

Symbolic constants are defined using a special keyword, #define. The #define is followed by at least one blank, the name of the constant, then at least one blank, and the string of characters. Upper case names are usually chosen for symbolic constants as a way to alert you that they are not the same as variables. Also, notice that a definition of a symbolic constant does not end with a semicolon. Let's take a look at a few definitions:

```
#define PI 3.14159
#define MAX pp
#define RSQUARED radius * radius
#define FORMULA PI * RSQUARED
```

You can use a symbolic constant anytime after its definition. When the constant's name is encountered, the compiler substitutes the replacement string. Here's some source code, using the above definitions, that shows what the compiler will actually process:

source code	expanded code
<pre>dim = radius * PI; cc = 5 * aa + MAX; printf("MAX = ",MAX); FORMULA;</pre>	<pre>dim = radius * 3.14159; cc = 5 * aa + pp; printf("MAX = ",pp); 3.14159 * radius * radius;</pre>

Did you notice that the characters MAX inside the string constant did not get expanded as a symbolic constant? Also take a look at the nested symbolic constant definitions of FORMULA and RSQUARED. First, the compiler substitutes PI * RSQUARED for FORMULA. It then examines the text again and determines that PI is a symbolic constant also. It then expands the code to 3.14159 * RSQUARED. When the second constant is encountered the code is expanded to 3.14159 * radius * radius;.

The following is Example 2.2 using a symbolic constant.

Example 2.3

```

#define PERCENT 0.03
main()          /* Example 2.3 */
{
    long total, stock, sales;
    float price, commission, cost, income;
    double profit;
    char type;

    /* input values to be used */
    printf("Please enter the following items ");
    printf("to calculate gross profit,\n net profit,");
    printf(" and sales commission on an item \n");
    printf("Enter the type of item (1 char.) : ");
    scanf("%c",&type);
    printf("Enter the number of items sold : ");
    scanf("%ld",&sales); /* items sold */
    printf("Enter the number of items on hand");
    printf(" at the beginning of the month :");
    scanf("%ld",&stock); /* items on hand */
    printf("Enter the current selling price : ");
    scanf("%f",&price); /* selling price */
    printf("Enter the cost of the item : ");
    scanf("%f",&cost); /* cost of item */

    /* calculate values */
    total = stock - sales;
    income = price * sales;
    profit = (price - cost) * sales;
    commission = profit * PERCENT;

    /* print answers */
    printf(
        "End of month total inventory = %ld of type %c\n",
        total,type);
    printf("On a gross income of %f profit = %f\n",
        income, profit);
    printf("Commissions at a rate of %f = %f\n",
        PERCENT, commission);
}

```

The symbolic constant PERCENT is used in this example to define the rate of the commission. By defining this as a symbolic constant, you will not have to search through the code to find all the occurrences of 0.03 in order to change its value. You simply would change the one line defining the symbolic constant, PERCENT. Notice that PERCENT was passed to printf as an argument in

order to print its value. Remember, if PERCENT had been used inside the format string it would have printed the word "PERCENT" and not its value.

You have been introduced to a few language features that can be used as tools to build useful C functions. Before you can effectively utilize these and the C operators you know about however, you must understand more about C expressions. For instance, what happens when you multiply an integer number times a floating point number? Or, in what order are the operators applied in this expression, $3 + 3 * 5$ (i.e. Is the answer 18 or 30?)

Now let's look at why these are problems. When several operators are in one expression, some of the operators are acted on before others. In the expression $3+3*5$ there are two operators, $*$ and $+$. The arithmetic operators $*$ (multiplication) and $/$ (division) are higher in precedence than $+$ (addition) or $-$ (subtraction). This means that multiplication and division are performed before addition and subtraction. Therefore, the expression would be evaluated as three times five (fifteen), plus three, resulting in a value of eighteen. The rule determining the order of evaluation is the operator with the highest precedence is evaluated first.

If however, you have the expression $3+2-4+6$, all of these operations are at the same precedence level. In this case, another rule is applied to the order of evaluation. This is known as associativity (grouping). All the arithmetic operators are binary (involving two operands) and group left to right. Therefore, $3+2-4+6$ is evaluated as three plus two which is five, subtract four which yields one, plus six yields seven. Let's look at another example of associativity, $a=b=c=2$. How is this expression grouped? The assignment operator groups right to left so it is evaluated as: $a=(b=(c=2))$, the variable c gets the value 2, then b gets the value of c (2), then a gets the value of b (2).

When expressions are evaluated, the precedence rule is applied first. Then the grouping rule is applied if there are operators of an equal precedence level in the expression. If you want to force a calculation to occur in a manner that does not fit the precedence or associativity rules, you can use parentheses. They are used in the same manner that parentheses are used in algebra: to force evaluation of the expression inside the innermost pair of parentheses, followed by the next innermost pair of parentheses, etc. Try to work these examples of C expressions and see if you understand both the above rules:

C expression	Equivalent to	Answer
$a=b=c=2*3-6*2$	$a=(b=(c=(2*3)-(6*2)))$	$a=-6, b=-6, c=-6$
$24\%5+7*4/7$	$(24\%5)+((7*4)/7)$	8
$(3+2)*4\%(6+3)$	$((3+2)*4)\%(6+3)$	2

The C language has quite a large number of operators. It also has 15 different levels of precedence. You will be introduced to the operators a few at a time. When the operator is introduced, a mention will be made of its precedence and associativity. However, the best habit to get into with long C

expressions is to use parentheses to force the order of evaluation.

You now can figure out what happens with operator precedence, but what happens when you execute the following that has a mixed type expression?

Example 2.4

```
main()          /* Example 2.4 */
{
    int a;

    a = 3.4 * 5;
    printf("a = %d \n",a);
}
```

Is the value of the variable `a`, seventeen or fifteen? In other words, was the multiplication performed as integer or floating point? To learn what happens, let's look at conversion of operands in C expressions.

When two operands in a binary operation are of different types, an automatic type conversion will occur. To understand the rules of this conversion process, you must understand that C types have a specific ordering. The lowest type is `char` and the highest type is `double`. The following table lists lowest type to highest type in a left to right order:

`char < int < unsigned < long < float < double`

The types of the two operands are compared and the lower type operand is converted to the higher type. The type of the result is the same as the higher type operand. In particular, the following conversion rules are applied in the order listed:

1. Any operand of `char` is converted to `int`.
2. Any operand of `float` is converted to `double`.
3. If either operand is `double`, then convert the other to `double`.
4. If either operand is `long`, then convert the other to `long`.
5. If either operand is `unsigned`, then convert the other to `unsigned`.

Another kind of type conversion takes place when the assignment operator is used. No matter what type expression appears on the right hand side of the assignment, the value is converted to the type of what appears on the left

hand side. Conversion of float to int truncates the fractional part of the number and long to int drops the excess high order bits. Integer to character drops excess high order bits also. Conversions such as these can lead to unexpected results unless you are aware that they are occurring. The following example shows some of the conversions that can take place.

Example 2.5

```
main()      /* Example 2.5 */
{
    char c1, c2, c3;
    int  i1, i2, i3;
    float f1, f2, f3;
    long d1;

    /* char converts to int */
    c1 = 'c';
    i1 = c1 - 'a' + 'A';
    c3 = i1;      /* truncate to character */
    printf("c1 = %c, i1 = %d, c3 = %c\n", c1,i1,c3);
    i1 = 321;
    c2 = i1;      /* convert integer to char */
    c3 = i1 + 1;  /* truncates value */
    printf("i1 = %d, c2 = %c, c3 = %c\n",
           i1,c2,c3);

    /* automatic conversion from int to float */
    f1 = 200;     /* converted to float */
    f2 = 350 * f1; /* 350 converted to float */
    /* 7 converted to float- result truncated */
    i3 = 3.4 * 7;
    /* 350 converted to float - result truncated */
    i1 = f3 = f1 / 350;
    printf("f1 = %f, f2 = %f,\n",f1,f2);
    printf("i3 = %d, f3 = %f, i1 = %d\n",
           i3,f3,i1);
    /* values produced in the following
    assume a 16 bit integer */
    d1 = 69631;   /* In hex 10FFF */
    i2 =d1;      /* truncates to 0FFF */
    printf("i2 = %d\n",i2);
}
```

The output of the program looks like this:

```

c1 = c, i1 = 67, c3 = C
i1 = 321, c2 = A, c3 = B
f1 = 200.000000, f2 = 70000.000000,
i3 = 23, f3 = 0.571429, i1 = 0
i2 = 4095

```

Take a look at this output carefully. In the statement,

```
i1 = c1 - 'a' + 'A';
```

the two character constants, 'a' and 'A' are converted to integer before the calculation is performed. You might notice this expression actually is converting the lowercase letter 'c' to uppercase 'C'.

The next part of the program,

```

i1 = 321;
c2 = i1;      /* convert integer to char */
c3 = i1 + 1;  /* truncates value */

```

shows an integer that is truncated to a character. Exactly which letters are printed as a result is not really important. This example shows that such conversion is possible. Usually, this occurs as a programming accident.

The next part of Example 2.5,

```

f1 = 200; /* converted to float */
f2 = 350 * f1; /* 350 converted to float */
/* 7 converted to float- result truncated */
i3 = 3.4 * 7;
/* 350 converted to float - result truncated */
i1 = f3 = f1 / 350;

```

shows some common floating point conversions. Notice that the expression $3.4 * 7$ is calculated in double floating point arithmetic and then truncated when the result is assigned to the integer `i3`. Also note in the next expression, the constant 350 is converted to floating point. The variable `f3` is assigned the floating point value(0.571429), but when the value is stored in `i1`, the floating point number is truncated to integer (0).

The conversion of long to int is shown in the following code:

```

d1 = 69631; /* In hex 10FFF */
i2 = d1; /* truncates to OFFF */

```

The exact representation in bits of this example is machine dependent. If an integer is 16 bits and a long integer is 32 bits in length, then the variable `d1` which has the value, 69631, expressed in bits is,

00000000000000010000111111111111. When the value of `d1` is converted to an integer, the high order bits are lost. The value of `i2` becomes, 0000111111111111, or 4095. Notice that this truncation of bits can drastically change the value of a variable!

This example just illustrates some of the rules we've discussed earlier. What is important to learn from it is that C may do some conversions automatically during calculations, sometimes making the results not what you intended.

The assignment and arithmetic operators that you've seen previously have some shorthand versions. These are known as assignment operators and they combine one arithmetic operator and the assignment operator, such as `+=`. For example, `x += 3` is the same as `x = x + 3`. There are other operators that can be used in this manner, but the following are the ones you've seen at this time:

Expression	Equivalent
<code>x += 4;</code>	<code>x = x + 4;</code>
<code>y -= 1;</code>	<code>y = y - 1;</code>
<code>z *= -4;</code>	<code>z = z * -4;</code>
<code>w /= 23;</code>	<code>w = w / 23;</code>
<code>x %= 2;</code>	<code>x = x % 2;</code>

One restriction applies to the assignment operators. The first operand must be a variable. For example, the following are not valid C statements:

```
0 += 3;
23 = a + 4;
1.2 + 5 = 6.2;
```

The increment and decrement operators also require that their operand be a variable and not a constant. The increment operator is used to add one to the operand, whereas the decrement operator subtracts one. Both are unary operators and therefore require only one operand. For example, `a++`, is equivalent to `a = a+1` and `a--` is equivalent to `a = a-1`.

In the statement, `bb = aa++`;, the value of `aa` is incremented by one after assigning the current value of `aa` to the variable `bb`. In this case `++` is a postfix operator, it increments the value of the variable `aa` after assigning the current value of the variable `aa` to the variable `bb`.

The increment and decrement operators can also be written prior to the operand. This is known as a prefix operator, the value of `aa` in this statement, `bb = --aa`;, is decremented before the assignment to the variable `bb`. The precedence of the increment and decrement operators are higher than anything you've seen before except the parentheses. For example:

The statement:

```
b = c + a++;
```

is equivalent to:

```
b = c+a; a = a+1;
```

The statement:

```
b = c + --a;
```

is equivalent to:

```
a = a-1; b = c+a;
```

This next example will show some assignment operators as well as increment and decrement operators.

Example 2.6

```
main()          /* Example 2.6 */
{
    int xx,ii;
    int yy,jj;

    jj = ii = 0;

    xx = jj++ + ++jj;
    printf("xx = %d, jj = %d, ii = %d\n",xx,jj,ii);

    yy = jj *= xx++ + ++ii + 3;
    printf("yy = %d, jj = %d, xx = %d, ii = %d\n",
        yy,jj,xx,ii);

    ii = ++ii;
    printf("ii = %d\n",ii);

    printf("1st --yy = %d, 2nd --yy = %d",--yy,--yy);
}
```

Output from Example 2.6

```
xx = 2, jj = 2, ii = 0
yy = 12, jj = 12, xx = 3, ii = 1
ii = 2
1st --yy = 11, 2nd --yy = 10
```

Take time to analyze the output from this example. Every expression shown has a side effect. That is, the increment/decrement operators change the

value of a variable during evaluation of the expression. But when is the value changed? Let's examine the statement:

```
xx = jj++ + ++jj;
```

The increment operators have the highest precedence and will therefore be evaluated first. The left operand of the addition(+) is first evaluated. The result is the current value of `jj`, which is zero. But then `jj` is incremented to the value one. The right operand of `+` is then evaluated. The variable `jj`, currently equal to one, is incremented to two. This is the result of the right operand. The result of the left operand (0) is then added to the result of the right operand (2). The value two is then assigned to `xx`. After this statement has finished, `jj` has the value two and `xx` has the value two. Also note the following expression:

```
yy = jj *= xx++ + ++ii + 3;
```

has five plus signs in a row. You must separate the symbols by blanks to make clear that you desire a post-incremented `xx` added to a pre-incremented `ii`. The compiler will try to take the longest string of characters that will form an operator. So, if you wrote `+++++`, the compiler would see `++`, then another `++`, followed by a `+` which is not the desired set of operations.

Now let's summarize what you've learned. This chapter introduced you to formatted input, symbolic constants, C data types, assignment operators, and the increment/decrement operators. You also should feel very familiar with C expressions including those that involve more than one operator and more than one data type. With this background, you are ready to learn about C control structures.

Chapter 3

Control Statements

So far, all the functions you've seen use sequential execution. That is, the function is entered, the statements in the function are executed one by one from top to bottom, and then the function is exited. But C has control statements that will let your program determine the flow of control (which statements will be executed next).

To use C control statements, you first need to learn about conditional tests. A conditional test is an expression that results in a true or false value. C has special operators for testing conditions between two operands, known as relational operators. These operators produce either a non-zero value or zero, depending on whether the relation tested is true or false. The relational operators include:

==	is equal to
!=	is not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

These operators can be used in any C expression. The following are examples of relational operators.

Example 3.1

```
main()
{
    int a,b,c;

    a = 3;
    /* b is assigned a non-zero value, true */
    b = a==3;
    /* c is assigned a zero value, false */
    c = a!=3;
    /* b == c will be false (0) */
    printf(" b == c should be zero(false),");
    printf(" the value is %d \n",b == c);
}
```

The statement `b = a==3` reads, "Assign the result of the test, is a equal to three, to the variable b". Consequently, the value of b is non-zero, true. The value of c in the statement `c = a!=3` is zero, because the expression `a!=3` is false. In the last statement, the expression `b == c` evaluates to 0 (false) since b is not equal to c.

Although the relational operators can be used in this way, the most common usage is in a C control statement. The simplest form of C control statement is the if statement. The format of the if statement looks like this:

```
if (expression)      /* If the expression is true */
    statement        /* execute this statement    */
```

The C reserve word, if, is followed by an expression enclosed in parentheses. If the expression results in a value of zero(false), the statement following the closing parenthesis is skipped. If the result of the expression is non-zero(true), the statement is executed. The next example makes use of the if statement and relational operators.

Example 3.2

```

main()                                /* Example 3.2 */
{
    int birth_yr, cur_yr;

    printf("Enter the year of your birth: ");
    scanf("%d",&birth_yr);
    if (birth_yr <= 0)
        printf(" Invalid year entered \n");
    printf("Enter the current year: ");
    scanf("%d",&cur_yr);
    if (cur_yr <= 0)
        printf(" Invalid year entered \n");
    printf(" Your current age = %d \n",cur_yr - birth_yr);
}

```

This example shows you a good way to format your C program. The indented statement after the if will be executed only if the condition is true. When you glance over this source code, it is clear which statements are conditionally executed.

The else statement provides a way to specify an alternate statement to execute when the expression of the "if" is 0 (false). The else statement is only valid after an if statement and looks like the following:

```

if (expression)
    statement_1    /* if expression is true execute this */
else
    statement_2    /* execute, if the expression is false*/

```

Specifically, the else statement works like this: If the expression evaluates to zero(false), then statement_1 is skipped and statement_2 is executed. If the expression is non-zero(true), the opposite occurs, statement_1 is executed and statement_2 is skipped. Also, note that if and else are two separate C statements and as such require semicolons. Take a look at this example:

```

if (a < 99)
    b = 3;        /* ; required at end of statement */
else
    b = 4;        /* ; required at end of statement */

```

Due to the semicolon before the else, this syntax(form) perhaps will look a little strange to PASCAL programmers. By using the if-else combination, you

can make Example program 3.2 much nicer. Why? Well currently, an invalid number can be entered for the birth year, then an error message is printed out and the program continues by asking for the current year. The program proceeds to calculate an age with the invalid input. By using the else statement, you can prevent the invalid calculation from happening. Take a look:

Example 3.3

```
main()                /* Example 3.3 */
{
    int birth_yr, cur_yr;

    printf("Enter the year of your birth: ");
    scanf("%d",&birth_yr);
    if (birth_yr <= 0)
        printf(" Invalid year entered \n");
    else {
        printf("Enter the current year: ");
        scanf("%d",&cur_yr);
        if (cur_yr <= 0)
            printf(" Invalid year entered \n");
        else printf(" Your current age = %d \n",
            cur_yr - birth_yr);
    }
}
```

The above example also illustrates the usage of the compound statement with the else statement. Any C statements between the left brace, "{", and the right brace, "}", are treated as if they were one C statement (remember the definition of a block?). Now the calculation for age is performed only if both dates are valid (greater than zero).

Next, let's look at the simplest C control loop, the while statement. The format of this statement is as follows:

```
while(expression)
    statement
```

The while reserve word must be followed by a parenthesized expression, just like the if statement. This while loop works by executing the statement over and over as long as the expression is true (non-zero). When the expression being tested is false (0), the statement is skipped and execution starts at the first statement following the while loop. The statement can be a simple C statement or a compound C statement enclosed by braces, {}.

Before taking a look at an example of the while loop, let's look at how to

initialize variables in declarations. A variable name may be followed by an equal sign and an initial value in a declaration. Variables are initialized to the constant value specified each time the function is entered. Later, you will see initialization of other data types. The following example contains initializers for the variables result and counter.

Example 3.4

```
main()          /* Example 3.4 */
{
    /*          y          */
    /* calculate x          */
    /*          */
    /* where x and y are integers and y >= 0 */

    int base, power;
    long result = 1;
    int counter = 0;

    printf("Enter the base number : ");
    scanf("%d",&base);
    printf("Enter the nth power to ");
    printf("which base will be raised:");
    scanf("%d",&power);

    while (counter++ < power)
        result = result * base;

    printf("The base of %d raised to",base);
    printf(" the %dth power is %ld\n",
        power,result);
}
```

In example 3.4, after you enter the base and power, the loop condition for the while is tested. While counter is less than power, the next statement is executed. Notice this example uses the post increment operator which you have seen before. After the loop condition is tested, the counter is incremented by one. Next, the variable result is multiplied by the base number. When the variable counter is equivalent to power the while loop is terminated and the final printf statement is executed.

Any of the C control statements can be used as the statement part of another control statement. In other words, an if statement can have another if statement as the part executed when the conditional expression is true. This is known as nesting the control statements. However, you must be careful to construct the expressions and statements so that the program does exactly what you expect.

Every else statement must be matched with a preceding if statement. When if-else statements are nested, the compiler assumes that the else statement is associated with the nearest unmatched if statement. Sometimes this makes your program behave in a way that you didn't intend. Consider the following examples. The intent is to assign the variable, result, with a value that is based on the value of the variable num such that:

```
result = 0 if num <= 0
result = 1 if 0 < num < 1000
result = 2 if num >= 1000
```

Nesting Examples

<pre>main() { int result=2,num; printf("Enter the number : "); scanf("%d",&num); if (num > 0) if (num < 1000) result = 1; else result = 0; /*result = 0 if num >= 1000*/ }</pre>		<pre>main() { int result=2,num; printf("Enter the number : "); scanf("%d",&num); if (num > 0) if (num < 1000) result = 1; else ; /*null statement*/ else result = 0; /*result = 0 only if num < 0*/ }</pre>
---	--	--

If you examine the code on the left, the variable result is set to 0 when num is greater than or equal to 1000. However, in the case on the right, result is set to 0 when num is equal to or less than zero. The program on the left is not correct because the else statement is matched with the second if statement. This happens even though the code was indented to show the else to match the first if statement. In the program on the right, an extra else statement was inserted to match the second if, so this program would execute as intended.

Instead of using the null statement after the else in the right hand example, you could use braces to clarify the nesting of the statements. This tells the compiler to treat all the C statements inside the braces as one statement. The else statement is then matched with the first if. The nesting example using braces would look like this:

Nesting using braces

```

if (num > 0) {
    if (num < 1000)
        result = 1;
}
else result = 0;

```

When testing two values for equality, watch out for the accidental use of equal (=) operator instead of the relational equality (==) operator. A while loop, in particular, may never terminate if the expression being tested has this problem. Take the next example and change the equality operator to an equal operator and see what happens.

Example 3.5

```

main()                                /* Example 3.5 */
{
    int sum = 0;
    int number, ok = 1;
    /* while never terminates if expression
       is ok = 1 */
    while (ok == 1) {
        printf("Enter a number to sum");
        printf("(zero will terminate): ");
        scanf("%d",&number);
        if (number == 0) ok = 0;
        else sum = sum + number;
    }
    printf("The total sum is %d\n",sum);
}

```

Do you see why the loop never terminates if the while expression is `ok = 1`? Even if the variable `ok` has been changed to zero inside the while loop, the result of the expression, `ok = 1`, is always non-zero(true). Consequently, the loop will never terminate.

Also notice in this example that the declaration for the variables `number` and `ok` has an initializer. This initializes the variable `ok` to 1. The variable `number` does not have an initial value.

Frequently you would like to test the results of evaluating two or more

relational operators. For example, is the variable `aa` equal to one and the variable `bb` less than two? The and operator (`&&`) and the or operator (`||`) which are known as logical operators accomplish this. The `&&` (and) operator is true only if both operands are true (not equal to zero). The `||` (or) operator is true if either one or both of the operands is true. Here's a list of how these operators work:

operand 1		operand 2	result of operation	
true	<code>&&</code>	true	true	
true	<code>&&</code>	false	false	<code>&&</code> (and) operator
false	<code>&&</code>	true	false	
false	<code>&&</code>	false	false	

operand 1		operand 2	result of operation	
true	<code> </code>	true	true	
true	<code> </code>	false	true	<code> </code> (or) operator
false	<code> </code>	true	true	
false	<code> </code>	false	false	

Some examples of logical expressions:

```
/* Is age greater than 5 AND less than 18? */
age > 5 && age < 18
/* is c a lower case OR an upper case letter? */
(c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')
```

The logical operators `&&` and `||` are binary operators and therefore require two operands. However, the `!` operator (not) is a unary operator and only requires one operand. It yields a non-zero (true) result when the operand is false and a zero (false) when the operand is true. Look carefully at these examples of the not operator because the results can be confusing:

```
!z /* true if z is false, equivalent to z == 0 */
!aa == 0 /* true if aa is true, a non-zero */
```

The logical and relational operators can be combined in a single expression. When combining relational and logical operators, the relational operators have the highest precedence and the `&&` operator has precedence over

the `||`. Be sure to check the rules of precedence or use parentheses if you are unsure which operation will be done first. Some expressions that are equivalent follow:

expression	equivalent
<code>a == b c < d</code>	<code>(a==b) (c < d)</code>
<code>bb > cc && dd == aa cc <= aa</code>	<code>((bb > cc) && (dd == aa)) (cc <= aa)</code>

The most common usage of logical operators is in conditional tests. The following example shows a logical expression that tests for alphabetic characters.

Example 3.6

```

main()                                /* Example 3.6*/
{
    char in = 'a';

    printf("Test for upper or lowercase letters,");
    printf(" terminate on non-alpha\n");
    while((in <= 'z' && in >= 'a') ||
           (in <= 'Z' && in >= 'A')){
        printf("Enter a character and");
        printf(" press the enter key: ");
        scanf("%c%c",&in);
        printf("Character read is %c\n",in);
    }
    printf("Program Terminated - ");
    printf("non-alphabetic character entered\n");
}

```

The test for alphabetic characters in the while loop expression is equivalent to the C library function `isalpha`. Notice the conversion specification used to read in each character. It looks like this: `%c%c`. This specification uses the suppression character, `*` to specify skipping over the next character. On each call to `scanf` two characters are read. The first is stored in variable `in`, while the second is discarded. If you did not add the `%c` to the specification string, then the carriage return would be read as an input character. This would cause an early termination of the while loop.

In writing loops, you frequently want to execute the loop for a predetermined number of times. To do this you would specify an initial value for the loop counter, a test expression to determine when the loop is done, and an increment or decrement for the loop counter to be applied at the end of the loop. Conveniently, C has a loop construct that will let you specify any or all of these three things. This is C's for loop. The format of the for

statement looks like this:

```
for ( expression1 ; expression2 ; expression3)
    statement    /* can be compound */
```

The first expression is used to initialize the loop counter variable. The second expression is the terminating condition test. The third expression is used to modify the loop counter. The for statement is also equivalent to the following C code:

```
expression1;          /* initialize counter */
while(expression2) {  /* test counter value */
    statement;        /* can be compound */
    expression3;      /* modify counter value */
}
```

Notice that the three parts of the for loop are separated by semicolons in this next example. It shows the for loop being used to sum five numbers.

Example 3.7

```
main()                /* Example 3.7 */
{
    int sum, count, num;
    printf("This program sums 5 integer numbers \n");
    sum = 0;
    for (count=0; count < 5; count++) {
        printf("Enter a number : ");
        scanf("%d",&num);
        sum += num;
    }
    printf("Total of the five numbers = %d\n",sum);
}
```

The variable count is initialized to zero, then count is compared to five. If count is less than 5 then the compound statement of the for loop is executed. Finally, the count is incremented and the test performed again. After the five numbers have been entered, the sum is printed out and the program ends. Notice that the first expression of the for, count=0, is executed only once.

Sometimes it would be nice to initialize two variables in the first expression of the for statement. This is accomplished using the comma operator. This operator takes two expressions and makes them appear as one to the compiler. The next example will show this operator.

The for statement, like the while statement, can be nested inside of other loops. Combined with the comma operator and nesting, the for statement can be a very powerful construct. Take a look at the example 3.7 again and see what it looks like with a comma operator in the initializer expression and with an extra outside loop added.

Example 3.8

```
#define TRUE 1
main()          /* Example 3.8 */
{
    int sum, count, num;    /* declarations */
    int go = TRUE;

    printf("This program sums 5 integer numbers \n");
    /* initialize go so that we fall through loop at least once */

    while (go) {
        for (sum = 0, count=0 ; count < 5; count++) {
            printf("Enter a number : ");
            scanf("%d",&num);
            sum += num;
        }
        printf("Total of the five numbers = %d\n",sum);
        printf(
            "Do you have any more sets of 5 numbers to sum?\n");
        printf("Enter 1 for yes and 0 for no: ");
        scanf("%d",&go);
    }
}
```

Be sure to note that the variable go must be initialized before it is used inside the while test. Otherwise, the program will not work properly.

There is one C looping structure you haven't seen, the do-while. This loop contains a conditional expression at the end of the body of the loop. This means that even if the conditional expression is false, the loop is executed at least once. The format of this statement is:

```
do                /* execute                */
    statement      /* this statement                */
while (expression) /* while this expression is true */
```

The following is Example 3.8, written using a do-while loop. Notice that you no longer have to make sure the variable go is initialized so that the loop is executed the first time.

Example 3.9

```
main()                                /* Example 3.9 */
{
    int sum, count, num;
    int go;
    printf("This program sums 5 integer numbers \n");
    do {
        for (sum = 0, count=0 ; count < 5; count++) {
            printf("Enter a number : ");
            scanf("%d",&num);
            sum += num;
        }
        printf("Total of the five numbers = %d\n",sum);
        printf("Do you have any more sets of 5");
        printf(" numbers to sum?\n");
        printf("Enter 1 for yes and 0 for no\n");
        scanf("%d",&go);
    } while (go);
}
```

You now should be familiar with the C control statements, if, while, and for. Also, you are hopefully able to write test conditions for these control statements using both logical and relational operators, in addition to using the arithmetic operators introduced earlier. In the next chapter, you will learn about C functions and then a few more operators will be introduced.

Chapter 4

Functions

In the previous chapters, you learned enough C to begin writing interesting functions. In this chapter, you will learn more about functions and how to write them. To start with, let's look at how the function call effects the flow of control in a C program.

When a function is called, the flow of control passes to the called function. Execution starts at the first statement in the function and continues until the function ends or until a return statement is encountered. The control of the program is then returned back to the calling function. Execution in the calling function resumes at the statement following the function call. The following example uses two functions in addition to the function main.

Example 4.1

```
main()
{   int    Score, Avg_Score, No_of_Students;
    long   Total_Score;
    prompt1();
    No_of_Students = 0;
    Total_Score = 0;
    prompt2();
    scanf("%d", &Score);
    while (Score != -1) {
        Total_Score = Total_Score + Score;
        ++No_of_Students;
        prompt2();
        scanf("%d", &Score);
    }
    printf("You Entered Scores for %d Students.\n", No_of_Students);
    If (No_of_Students > 0) {
        Avg_Score = Total_Score/No_of_Students;
        printf("Average Score was %d.", Avg_Score);
    }
}
```

```

prompt1()
{
    Printf("*** Program to Compute Average Test Score ***\n");
    Printf("    A Test Score of -1 Terminates Input\n");
}

prompt2()
{
    printf("Enter Test Score: ");
}

```

This example reads integer values and prints four values per line. After six lines are output, a new heading and the next set of values are printed. The functions `newline` and `newset` do not require any arguments and they do not return any defined values.

A C function call is an expression that results in the value returned by the called function. You can call a function from any place that an expression is legal. The following C code segments show some examples:

```

call_it();           /* discard function result */
result = call_it(); /* save function result   */
while (result = call_it()); /* call until 0 is returned*/
if (a == (result = call_it())) /* compare a to the result */

```

These examples require some further explanation. The first statement is a simple function call. The value it returns is discarded. However, in the second statement, the value returned by `call_it` is saved in the variable `result`. The third statement not only saves the variable `result`, it uses the value of `result` as the `while` statement conditional expression. The last statement assigns the returned value of the function call to `result`, then tests to see if that value is equal to the variable `a`.

The value returned by a function is defined by the `return` statement. It has the following form:

```
return ( expression );
```

Execution of this statement returns control back to the calling function and the value of the expression is the value returned. The expression is optional. A `return` statement by itself will return an undefined value. If no `return` statement is present in a function, then control is returned to the caller when the closing right brace of the function is reached. In this case, the value returned is also undefined.

To pass values into a function, arguments are used. You've seen how values are passed in as arguments. What you need to know now is how to define a

function with arguments. All argument declarations are specified after the function header and before the body of the function.

Here is the standard C library function, `isalpha`:

```
isalpha(c)          /* returns 1 if c is a letter */
int    c;           /* else returns 0          */
{
    if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z') return 1;
    else return 0;
}
```

Notice the declaration for the argument `c` before the first left brace. This declaration is not required because the argument's type defaults to integer. However, it is a good practice to put in argument declarations so that you will not forget them when the arguments are non-integer. Also take a look at the return statements inside the `isalpha` function. The return statements define the values that are returned to the caller.

Now let's look at how the control of execution is passed from function to function. Example 4.2 shows this change of execution control. Each function in this example will be discussed separately. Then the complete listing will be shown so that you can try it on your computer.

Main function

```
main()              /* This starts example 4.2 */
{
    int i,sq,cu;

    printf("This function prints the square");
    printf(" and the cube of a number\n");
    printf("Enter the number now: ");
    scanf("%d",&i);
    printf("\nThe value entered = %d\n",i);
    sq = square(i);
    printf("\nThe value squared = %d\n",sq);
    cu = cube(i);
    printf("\nThe value cubed = %d\n",cu);
}
```

The first function is the main function. It calls the `square` and `cube` functions (besides `printf` to print the values returned). Later you will see in the definition of `square`, a value is returned. The main function stores the values returned from `square` as well as `cube`, even though C does not require you to utilize or store the value returned from a function.

Square function

```
square(ii)
int ii;
{
    return(ii * ii);
}
```

The square function returns the square of the value passed as the argument. The squared value is not stored in a local variable. It is passed directly back to the calling function using the return statement.

Cube function

```
cube(iii)
int iii;
{
    return(iii * square(iii));
}
```

The cube function returns the cube of the passed argument. This result is calculated by getting the square of the argument and multiplying. Notice, the square function has been called from this function as well as from the main function. Any C function may call any other C function. Now take a look at this example listed altogether.

Example 4.2

```
    /* This is a complete listing of example 4.2 */
main()
{
    int i,sq,cu;

    printf("This function prints the square");
    printf(" and the cube of a number\n");
    printf("Enter the number now: ");
    scanf("%d",&i);
    printf("\nThe value entered = %d\n",i);
    sq = square(i);
    printf("\nThe value squared = %d\n",sq);
    cu = cube(i);
    printf("\nThe value cubed = %d\n",cu);
}

square(ii)
int ii;
{
    return(ii * ii);
}

cube(iii)
int iii;
{
    return(iii * square(iii));
}
```

Program Output from Example 4.2

This function prints the square and the cube of a number
Enter the number now: 13

The value entered = 13

The value squared = 169

The value cubed = 2197

Take a look at the above output from this program and see if you can trace

the flow of control. Notice that the main function calls the function cube that in turn calls the function square. The value returned by square is then used in the calculation of the value to be returned by cube.

The functions in Example 4.2 all return values of type int. The function header "square()" is equivalent to "int square()". A function can be defined to return a value of some other type by preceding the function name with a type keyword. For example:

```
double sin(angle) {...}    /* double precision result */

/* The declaration is read as "the function sin
   returning type double".*/

long bignum(x) { ..... } /* long integer result */

/* The declaration is read as "the function bignum
   returning type long". */
```

When the arguments passed to a function are not integer, they must be declared before the body of the function. Now look at these examples of function headers and argument declarations.

```
double cos(angle)    /* function returning double */
double angle;       /* argument passed is double */
{...}

int putdbl(value)   /* function returning int    */
double value;      /* argument passed is double */
{...}
```

The conversion rules for expressions you saw earlier also apply to the arguments passed to a function. Char type operands are converted to int and floating point operands are converted to double at the time of the function call. Therefore, the arguments of a function should never be declared as type char or float. You should instead use int or double. Otherwise, you will get unexpected results.

A function returns a value of the type specified in the function header. Even if the expression of a return statement is a different type, the value is converted to the correct type before being returned. The following is an example of a definition of a double precision function:

```
double cir_area(radius)
double radius;
{
    return(3.14159 * radius * radius);
}
```

You now have a function declared that returns a value of type double, but to use this function more has to be done. Recall that a function is assumed to be "function returning int". You now have to tell the function that calls `cir_area` that the value returned is of the type double. This is accomplished with a declaration like the following:

```
double cir_area(); /* function returning double */
```

Every function that uses `cir_area` must contain the above declaration. This tells the calling function that the value returned is double. A common C programming error is to forget such declarations, leaving the compiler to assume that `cir_area` returns an integer. This will usually cause an abnormal termination of the program. It is a very good practice to declare all functions used so that these problems do not occur. The next example shows the definition and declaration of the function `cir_area` which calculates the area of a circle. Notice in the declarations that the types agree.

Example 4.3

```

                                     /* Example 4.3 */
double cir_area(radius) /* function definition */
double radius;
{
    return(3.14159 * radius * radius);
}
main()
{
    double cir_area(); /* function declaration */
    double radius;
    double area;
    printf("This program calculates the");
    printf(" area of a circle\n");
    printf("Enter the radius of the circle: ");
    scanf("%lf",&radius);
    area = cir_area(radius);
    printf(" The area = %f ",area);
}

```

One thing you should note, this example uses the conversion specification of `%lf` to read the double precision variable, `radius`. Without the correct conversion specification, the value of `radius` would have been unpredictable.

There is a special C type that has not been covered, the type void. It is used to declare functions that do not return a value. This tells the compiler not to reserve any space for a function result and to ignore any values produced by return statements. The type `void` is used just like any other

function returning a type other than integer. The function must be declared as void in both the header of the definition and in the declarations of any function that references it. The following program illustrates the use of type void.

Example 4.4

```
#define TRUE 1
main()                                /* Example 4.4 */
{
/* print a number in one of several formats */
  int go = TRUE,type,dnum;
  float fnum;
  void fltout(), intout();

  printf("Print a number as ");
  printf("binary or hexadecimal\n");
  while(go) {
    printf("Is your number floating point ?\n");
    printf("Enter 0 for no, ");
    printf("anything else for yes: ");
    scanf("%d",&type);
    if(type) {
      printf("Enter your floating number: ");
      scanf("%f",&fnum);
      fltout(fnum);
    }
    else {
      printf("Enter your integer number : ");
      scanf("%d",&dnum);
      intout(dnum,'D');
      intout(dnum,'B');
      intout(dnum,'H');
    }
    printf("Do you wish to enter another number?\n");
    printf("Enter 0 for no, anything else for yes\n");
    scanf("%d",&go);
  }
}

void fltout(dnum)
double dnum;
{
  float fnum;

  printf("\nFloating Point number: %f\n",dnum);
  printf("in e format: %e\n",dnum);
}

```

```

void intout(inum,type)
int inum,type;
{
    void binary();

    if (inum < 0) {
        printf("Only positive numbers accepted\n");
        inum = - inum;
    }
    if (type == 'B') {
        printf("Binary: ");
        binary(inum);
        printf("\n");
    }
    else if (type == 'H')
        printf("Hex: %x\n",inum);
    else
        printf("Decimal: %d\n",inum);
}

void binary(inum)
int inum;
{
    if (inum > 1) binary(inum/2);
    printf("%d",inum%2);
}

```

In the main function of this example, notice that the functions, `fltout` and `intout`, are declared to be of type `void`. This declaration is required since both functions are defined as type `void`.

Also notice that in the function `fltout`, the argument `dnum` is declared to be `double`. The argument `fnum` passed to it from `main` is a `float`. Do you remember that C automatically converts `float` in an expression to `double`? The usage of `fnum` as an argument is an expression. Therefore it is converted to `double`. The argument in `fltout` must be declared `double` for it to behave correctly.

You'll see a similar situation in the function `intout`. In the main function, `intout`'s second argument is a character constant. However, in the definition of `intout` the second argument is declared to be `integer`. Recall that any character operand in an expression is converted to `integer`. Also take note of the fact that `intout` also has to declare the function `binary` as type `void`.

The function `fltout` uses a new conversion specification, `%e`. This specification is just a different way of printing out a floating point number. What is printed is the floating point number in scientific notation (a number times a power of ten). Also used is the conversion specification, `%x`. It is used to print an integer as an unsigned hexadecimal number.

You have now covered most of the basic control structures and data types of C. You should feel comfortable with all the topics covered up to this point. If not, please review the material presented. More sophisticated programming techniques and data structures will be covered in the next chapters and these topics by necessity build on the basic information already presented.

Chapter 5

More I/O and Control Statements

You are now ready to learn about character I/O and some other control structures. In order to fully comprehend character I/O, you will need to learn about C include files. Then you will be introduced at various times to the definitions that are included from the standard header file, `stdio`. The standard header file must be included if your program performs input or output using any functions other than `scanf` or `printf`.

The `#include` statement allows us to include in our program, C source from another file. The `#` must be followed immediately by the word `include`. Next is the file name enclosed by " " (double quotes). The format of this statement is:

```
#include "filename" /* valid file on your system */
```

The filename must be the proper syntax for the system on which you are running. See the Systems Implementation Manual for details. When the compiler sees the `#include` statement, it suspends compiling in the current file and starts compiling source from the included file. When the included file has been compiled, the compiler continues compiling at the statement following the `#include`. The file included can also contain `#include` statements.

To set up this example of multiple include statements you will have to put the following sections of code in separate files and name the files in the proper syntax for the operating system you are using. The example will use `FILE1` and `FILE2`. Enter this first section as `FILE1`:

```
FILE1
/* _____ */
/* common preprocessor definitions from FILE1 */
#define FALSE 0
#define TRUE 1
#define YES 1
#define NULL '\0'
#define ALL 1
```

And this second section needs to be in `FILE2`:

```

FILE2
/* _____ */
/* common declarations from FILE2 */
int  one,two,three;
char a,b,c;
float flt1,flt2;

```

This is the main function that will include the previous files:

Example 5.1

```

#include "FILE1"
main()
{
    #include "FILE2"
    a = 'a';
    one = 1;
    flt1 = 3.567;
    printf("This example shows include files only\n");
}

```

Take a look at the compilation listing from this example. You will see all three files. The include feature is very handy for automatically adding a set of declarations to every compile you do. In fact, most C programmers will place a #include for the standard header file in every C compilation, regardless of whether the definitions are needed.

Now, let's look at the character I/O functions. Recall that a C program automatically opens three files: stdin, stdout, and stderr. The scanf and printf functions introduced earlier used the file stdin and stdout by default. You will now learn about the standard functions, getc and putc, which require you to specify the file name to be used.

The function getc returns one character from the file specified as the argument. Right now the only file you know how to access for input is stdin. The call to getc looks like the following:

```
onechar = getc(stdin);
```

If the file has no more characters, the special value, EOF (end of file) is returned. This value is system dependent and is defined as a symbolic constant in the standard header file supplied with your system. The result of the function call, getc, should be checked in order to detect if EOF was returned or your program could get unexpected results.

The function to output a character is putc. Its arguments are the

character and the file to which the character will be sent. Stdout and stderr are the only two files that you know about right now for output. Later you will learn how to open, close and manipulate other files. When you use a `putc` function call, it looks like this:

```
putc(c,stdout);
```

The variables `stdout`, `stdin`, and `stderr` are addresses that are defined in the standard header file, `stdio`. They are really pointers to the files. However, this tutorial has not yet covered how to declare pointers, so for right now think of them just as symbolic constants. The following is a simple file copy program using `getc` and `putc`:

Example 5.2

```
#include "stdio"           /* standard header */
main ()                   /* Example 5.2 */
{
    /* this program reads characters
       from stdin and writes them
       to stdout */

    int charin;

    /* See the Systems Implementation Manual for how
       to generate EOF from the keyboard */

    while ( (charin = getc(stdin)) != EOF )
        putc(charin, stdout);
}
```

When this program runs, it will read characters from the keyboard and output them to the screen. You may also redirect `stdin` and `stdout` to other devices or to a disk file as mentioned earlier in the tutorial. The character you type will be displayed once as an echo and once as an output from `putc`. Consequently, each line of input will be displayed twice. Consider the following example:

```
abc
abc
jklm
jklm
This is being entered
This is being entered
^Z
```

The first line is the characters you entered followed by a carriage return (generated by an enter or return key). The second line was printed by the `putc` function. The character you must enter at the keyboard for EOF is system dependent. It is shown in this example as control-Z, `^Z`. Please check the Systems Implementation Manual for which ASCII character is used for EOF.

The most commonly used character I/O functions are `getchar` and `putchar`. `Getchar` inputs a character from `stdin` and `putchar` outputs a character to `stdout`. The way they are called looks like this:

```
onechar = getchar();
putchar(onechar);
```

Anytime `getchar` or `putchar` is used, the standard header file must be included in the compilation. The compiler actually substitutes calls to the functions, `getc` and `putc`, when `getchar` and `putchar` are used. This is accomplished by something called macros. These will be covered in the tutorial later. These macros are defined in the standard header file. Now, lets see example 5.2 using `getchar` and `putchar`:

Example 5.3

```
#include "stdio"           /* standard header */
main ()                   /* Example 5.3 */
{
    /* this program copies a file from stdin to stdout */

    int charin;

    while ( (charin = getchar()) != EOF )
        putchar(charin);
}
```

This program is equivalent to the previous version. Notice that the include statement is required because the definitions for `getchar` and `putchar` are in the standard header file. Another thing to note about this example is the declaration of `charin` as an `int`, rather than `char`. The variable `charin` holds the character read by the function `getchar`. The `getchar` function actually returns a value of type `int`. If you declare `charin` as `char`, the value returned from `getchar` will be truncated according to the conversion rules discussed earlier. The result is that the character variable's value is always positive. This will not properly compare to EOF, a negative value. Thus, this conversion will cause the program to loop indefinitely. To avoid problems like this, always declare a variable as integer if it is to hold the value returned by `getchar` or `getc`.

Now you're ready to take a look at some of C's other control statements.

In using C control statements, sometimes you will want to end the looping before the normal loop termination. One way to do this is using the `break` statement. This causes a jump to the end of whatever C loop is currently being executed.

The `break` statement can be used in a `while`, `do-while`, or `for` statement. In all cases, it terminates the loop. If the `break` statement is inside nested loops, it applies only to the most immediately enclosing loop. Take a look at this program to count occurrences of digits, alphabetic characters, and whitespace.

Example 5.4a

```
#include "stdio"
main()                /* Example 5.4a */
{
    int count = 0, charin;
    int digit = 0, other = 0;
    int alpha = 0, space = 0;

    printf("\nThis program will count ");
    printf("occurrences of digits,\n");
    printf("alphabetic characters, and whitespace\n");
    printf("Enter EOF to terminate the program\n");
    printf("Reading input.....\n");
    for(count=0;;++count) {
        charin = getchar();
        if (isalpha(charin)) alpha++;
        else if (isdigit(charin)) digit++;
        else if (isspace(charin)) space++;
        else if (charin == EOF) break;
        else other++;
    }
    printf("\nSUMMARY\n");
    printf("\nThere were %d digits entered\n",digit);
    printf("Along with %d alphabetic characters,\n",alpha);
    printf(" %d whitespace characters,\n",space);
    printf("and %d other characters.\n",other);
    printf("For a total of %d characters read.\n",count);
}
```

This example shows the use of a `break` statement. The `for` loop termination expression is null (permanently false). On each iteration of the loop, this expression is tested for a true value, but the result is always false. The loop is terminated only by entering the EOF character which causes the `break` statement to be executed. When this happens, the summary is printed.

This example also makes use of three functions from the standard library,

isalpha, isdigit and isspace. These functions all perform character tests. Isalpha returns non-zero(TRUE) if the character is alphabetic. Likewise, isdigit returns non-zero(TRUE) if the character is a digit and isspace returns non-zero if the character is a blank, tab or newline('\n'). All three functions return a zero(FALSE), if the test fails.

Another way of changing the flow of control is to use the goto statement. In order to use the goto statement you must have a labeled statement. Then, the goto statement can branch to that label.

The label is a C identifier followed by a colon. It may appear by itself or on the same line as another C statement. If the label appears by itself on a line, the very next C statement is the one which is labeled.

An example of the format of these statements can be seen here:

Example 5.4b

```
#include "stdio"
main()                /* Example 5.4b */
{
    int count = 0, charin;
    int digit = 0, other = 0;
    int alpha = 0, space = 0;

    printf("\nThis program will count ");
    printf("occurrences of digits,\n");
    printf("alphabetic characters, and whitespace\n");
    printf("Enter EOF to terminate the program\n");
    printf("Reading input.....\n");
    for(count=0; ++count) {
        charin = getchar();
        if (isalpha(charin)) alpha++;
        else if (isdigit(charin)) digit++;
        else if (isspace(charin)) space++;
        else if (charin == EOF) goto summarize;
        else other++;
    }
    summarize:
    printf("\nSUMMARY\n");
    printf("\nThere were %d digits entered\n", digit);
    printf("Along with %d alphabetic characters,\n", alpha);
    printf(" %d whitespace characters,\n", space);
    printf("and %d other characters.\n", other);
    printf("For a total of %d characters read.\n", count);
}
```

Note this program will behave exactly like example 5.4a. The break statement in Example 5.4 is equivalent to executing a goto summarize;. This is the only example you will see in the tutorial of the goto statement. You will find C's control structures expressive enough that goto statements will be unnecessary. The labeled statement will be seen again shortly in the discussion of the switch statement.

In C you can also continue the next iteration of a loop. This is accomplished by the continue statement. It is similar to the break statement in that it causes a jump in the execution of the loop. It looks like this:

```
continue;
```

While the break statement terminates a loop, a continue statement continues the loop at the next iteration. In the while and do-while loops, execution continues at the conditional expression for the loop. In a for loop, execution continues at the third expression of the for. In the previous example, the third expression of the for is ++count.

The following example illustrates the use of the continue statement. The program reads characters from stdin, converts all tab characters to a sequence of eight blanks, and writes the characters to stdout.

Example 5.5

```
#define TAB 8          /* Example 5.5 */
#include "stdio"
main ()
{
    /* Convert tab characters to blanks */
    int c,i;          /* The tab character is generated
                       from the keyboard by <CTRL I> */
    while((c = getchar()) != EOF) {
        if ( c == '\t' ) {
            for ( i = 0; i < TAB; i++) putchar(' ');
            continue;
        }
        putchar(c);
    }
}
```

In the previous program, '\t' represents the tab character. Each time a tab character is read, the for loop outputs eight blank characters and then the continue statement causes a branch back to the while expression. For tab characters, the statement putchar(c); is not executed.

Now take a look at the C conditional expression operator, ? :. It operates like the if-else pair of C statements. This operator is the only C operator

that requires three operands. The first operand is the conditional test expression. If this expression is true, then the second expression is evaluated, otherwise the third expression is evaluated. This operator changes the flow of control just like the if-else statements do. But, it is an expression and as such, it generates a value which can be used like any other expression value. The following is a simple example:

```
printf(" The largest number is %d\n",(a>b) ? a : b);
```

This is equivalent to the following....

```
if (a>b)
    printf("The largest number is %d\n",a);
else
    printf("The largest number is %d\n",b);
```

As you can see, the statements using the ? : operator can be written in a lot less space. The following example uses the ? : operator :

Example 5.6

```
main ()                /* Example 5.6 */
{
    int largest, il, i2;

    printf("Enter 2 integer numbers: ");
    scanf("%d%d",&il,&i2);
    largest = il > i2 ? il : i2;
    printf("The largest number is %d",largest);
}
```

This example uses the ? : operator to set the variable largest to the correct value. The second and third expressions used with the ? : operator do not have to be simple variables such as il or i2, but can be complex expressions. However, with more complex expressions you should use parentheses to make the parts of the ? : operator clearer.

The last topic to be covered in this chapter is the most sophisticated of all the control structures, the switch statement. It is a way to transfer control like a series of if-else statements, except it uses only one test expression.

Now, let's find out how a switch statement could replace a series of if-else statements. The switch statement has an expression to evaluate. The result of the expression is compared to the constant values specified in case labels. These case labels specify where to start executing if a match is made. There can be a default label, to specify where to go if none of the

other cases is matched. The format of the switch statement looks like the following:

```

switch(expression) {
    case label_1: statement;
                statement;
                . . .
    case label_2: statement;
                statement;
                . . .
    case label_n: statement;
                statement;
                . . .
    default:    statement;
                statement;
                . . .
}

```

The case label must be a constant of any basic data type except float or double. When the switch statement matches the value of the expression and a case label, the flow of control is transferred to the statement following the matching label. When there is no match and no default label is specified, then the statement following the switch statement is executed, thereby skipping all statements associated with the case labels. Let's take a look at an example using a switch statement.

Example 5.7

```

#include "stdio"
main()          /* Example 5.7 */
{
    int digit = 0, other = 0, space = 0;
    int charin;

    printf("\nThis program will count ");
    printf("occurrences of digits,\n");
    printf("whitespace and other characters\n");
    printf("Enter EOF to terminate the program\n");
    printf("Reading input ..... \n");
    while((charin = getchar()) != EOF)
        switch(charin) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':

```

```

        case '6':
        case '7':
        case '8':
        case '9': digit++;
                    break;        /* exit switch */
        case '\t':
        case '\n':
        case ' ': space++;
                    break;        /* exit switch */
        default: other++;
                    break;        /* exit switch */
    }
    printf("\n\n SUMMARY\n");
    printf("There were %d digits entered\n", digit);
    printf("Along with %d whitespace characters\n",
        space);
    printf(" and %d other characters.\n", other);
}

```

In this example, multiple case labels are used to cause transfer to the same statement (i.e. if a digit is read, then the variable `digit` is incremented). Note that the `break` statement is used after each of the counters. Before, you saw that the `break` statement could be used to terminate a loop. The `break` statement may also be used to branch out of a switch statement. In the example, the switch statement is inside a while loop. However, the `break` statements are inside the switch statement. Therefore, they have no effect on the while loop. The `break` statement, inside a switch statement, causes a branch to the end of the switch statement. They are necessary in the previous example. Otherwise, you would have all three variables incremented when a digit is read, and the last two variables incremented when a whitespace character is read.

The following example further illustrates the use of the switch statement.

Example 5.8

```

#define DAYPHCORR 10;                /* Example 5.8 */
#define LENCYCLE 28.3;
main()    /* Same as Pascal tutorial 6.2 */
{
    int daynumber, intphase;
    int startphase, phase, month, day, year;
    double realphase, phasecorrection;

    printf(" *** Lunar Phase calculation program ***\n");
    printf(" Enter the month, day, and year:");
    scanf("%d,%d,%d ", &month, &day, &year);
    startphase = ((year - 78) * 365) + DAYPHCORR;

```

```
switch(month) {
    case 1: daynumber = 1;
           break;
    case 2: daynumber = 32;
           break;
    case 3: daynumber = 60;
           break;
    case 4: daynumber = 91;
           break;
    case 5: daynumber = 121;
           break;
    case 6: daynumber = 152;
           break;
    case 7: daynumber = 182;
           break;
    case 8: daynumber = 213;
           break;
    case 9: daynumber = 243;
           break;
    case 10: daynumber = 274;
            break;
    case 11: daynumber = 304;
            break;
    case 12: daynumber = 334;
            break;
}

startphase += daynumber + day;
realphase = startphase / LENCYCLE;
intphase = realphase;      /* truncate the result */
realphase -= intphase;
phase = realphase * LENCYCLE;
switch (phase){
    case 1: case 2: case 3: case 4: case 5: case 6: case 7:
        printf("The moon is in its first quarter.\n");
        break;
    case 15: case 16: case 17: case 18: case 19:
    case 20: case 21:
        printf("The moon is in its third quarter.\n");
        break;
    /* labels don't have to be in order */
    case 8: case 9: case 10: case 11: case 12:
    case 13: case 14:
        printf("The moon is in its second quarter.\n");
        break;
    case 22: case 23: case 24: case 25: case 26:
    case 27: case 28:
        printf("The moon is in its fourth quarter.\n");
        break;
}
}
```

This program computes the current phase of the moon. It is based on knowing the phase of the moon at some previous date (day and year). The variable `startphase` is the number of days passed since that date. The number of days is then divided by the lunar cycle length. Notice that several arithmetic expressions involve mixing integer with double variable types.

This chapter has introduced you to character I/O and a number of the other control statements. Also covered was the use of include files. Next, you will be looking at the use of these control statements with pointers and arrays.

Chapter 6

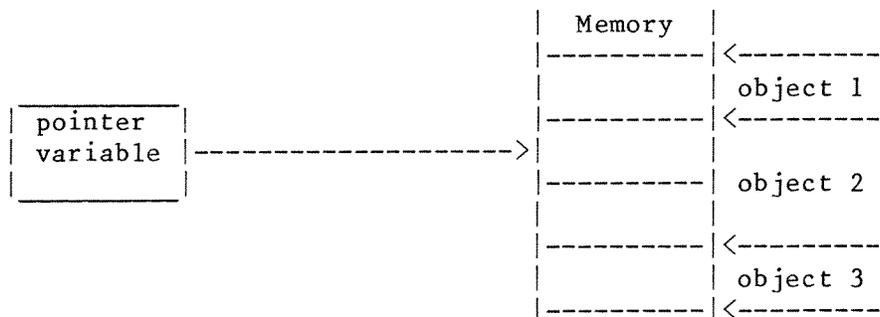
Pointers and Arrays

Pointers are used extensively in C. They correspond to the address of some data object. You were exposed to pointers when `scanf`'s arguments were described earlier. The address of operator (`&`) was used to obtain a pointer to a variable. Now, you will learn how to declare and use pointer variables.

Pointer variables are declared by placing an asterisk in front of the variable name. This declares the variable as a pointer to a value of the specified data type. Some declarations of pointers follow:

```
char *ptr1;           /* ptr1 is pointer to char */
float *ptr2, realnum; /* ptr2 is pointer to float */
int  alpha,*beta;    /* beta is pointer to int  */
```

The pointer data type always reserves the same amount of space in memory, just enough to hold a machine address (usually the same length as `int`). The declaration is merely a reservation of space and does not initialize the pointer to any address. Therefore, if you use a pointer before setting it to an appropriate address, then chances are your program will not behave the way you expect. To use a pointer in an expression, the unary operator for indirection, `*`, must be used. This operator interprets the operand as the address of the data you really want to use. It has also been referred to as the contents-of operator, because the value produced is the contents_of what is pointed to in the operand.



A pointer variable contains the address of an object in memory. In other words, it points to a particular location in memory. The size (in bytes) of an object depends on its data type. The type of the pointer variable determines the size of the object.

Example 6.1

```

main()                /* Example 6.1 */
{
    int *ptr1, xxx;
    int old, new;
    int *savptr;

    xxx = 9;
    ptr1 = &xxx; /* ptr1 contains address of xxx */
    printf("ptr1 = %x \n",ptr1);

    /* get the contents of what is pointed to by ptr1 */
    old = *ptr1; /* old now has the value of xxx */
    printf("old = %d \n",old);

    new = *ptr1 + 1; /* new contains xxx + 1 */
    printf("new = %d\n",new);

    savptr = ptr1; /* save the old address */
    printf("savptr = %x\n",savptr);

    ptr1 = &new; /* make ptr1 point to new */
    printf("ptr1 = %x\n",ptr1);

    old = *ptr1 + 1; /* an updated old value */
    printf("old = %d\n",old);
}

```

The output from this example follows:

ptr1 = AOCA		Value is address
old = 9		Old has the same value as xxx
new = 10		New is value of xxx + 1
savptr = AOCA		Pointers can be assigned
ptr1 = AOCE		Set ptr1 to new address (address of new)
old = 11		Now old has value of new + 1

The example uses the format conversion specification, %x. This prints the value of an integer variable in hexadecimal format. Since a memory address is normally represented in hexadecimal format, the %x specification should be used when printing the value of a pointer variable.

Recall from Chapter 3 that we can use the & operator to pass a variable by reference to a function. A variable must be passed by reference if the function is to alter the value of the variable. Inside the function

definition, the arguments of the function must be declared as pointers when passing values by reference. Consider the following example.

Example 6.2

```
main()                /* Example 6.2 */
{
    /* function returns value through arguments */
    int var1, var2;

    var1 = 25;
    var2 = 99;
    printf("Variables before swap,");
    printf("var1 = %d , var2 = %d\n",var1,var2);

    swap(&var1,&var2); /* swap these variables */
    printf("Variables after swap,");
    printf("var1 = %d, var2 = %d\n",var1,var2);
}

swap(ptr1,ptr2) /* function swap */
{
    int *ptr1,*ptr2; /* arguments passed by reference */
    {
        int temp;
        temp = *ptr1; /* save value of 1st argument */
        *ptr1 = *ptr2; /* 1st argument gets value of 2nd*/
        *ptr2 = temp; /* 2nd argument gets value of 1st*/
    }
}
```

In this example, you see how the * operator is used. The arguments of swap are both pointers to integer values. The statement

```
(*ptr1 = *ptr2)
```

is read, "Contents of ptr1 is assigned the contents of ptr2".

Another data type of C is the array. It is a set of data items in an ordered sequence, but it is identified by only one name. The items, called elements of the array, must all be of the same data type. The declaration of an array specifies the number of elements and the array's name. Declarations look like the following.

```
int digit[10];        /* digit is an array of 10 integers */
char letter[26];     /* letter is an array of 26 characters */
float number[100];   /* number is an array of 100 floats */
```

To reference an individual element of an array, you use the array name

followed by an open square bracket, [, an integer constant or integer variable (known as the subscript), and a closing square bracket,]. The following are examples.

```
digit[7]    letters[13]    numbers[2]
```

The elements of an array are numbered beginning with 0, so the array `digit` above has 10 elements numbered like the following:

```
digit[0],digit[1],digit[2].....digit[8],digit[9].
```

The beginning of the array (base) is the address of the first data item. The subscript is the index from the base item. Thus the first element, `digit[0]`, is 0 items from the base address and the second element, `digit[1]`, is 1 item from the base address, etc. Great care should be taken to have a valid subscript. The C language has no checks on the bounds of an array, so in the example above, `digit[10]` and `digit[255]` are both syntactically valid, but would produce unexpected results (random data from memory would be referenced).

The following example will use an array to keep track of the occurrences of the characters '0' through '9' in the program's input. First, the array is initialized to zeros because the elements will be used as counters. If the character read is a digit, then the appropriate array element is incremented (i.e. If the character is a '0' then `digit_count[0]` is incremented, for a '1' then `digit_count[1]` is incremented, etc.).

Example 6.3

```

                                /* Example 6.3 */
#define MAX 10                    /* bound of the array */
#include "stdio"                  /* standard header file */
main()
{
/* Program counts occurrences of numbers in input */

    int digit_count[MAX], i;
    int char_in; /* What happens if this is type char? */

    /* initialize the array to zeros */

    for(i = 0; i < MAX; i++)
        digit_count[i] = 0;

    /* find digits in input and count occurrences */

```

```

while ( (char_in = getchar()) != EOF) {
    if (char_in >= '0' && char_in <= '9')
        digit_count[char_in - '0']++;
}

/* print out information collected */

for(i = 0; i < MAX; i++)
    printf(
        "There are %d occurrences of %d in the input\n",
        digit_count[i],i);
}

```

Notice the expression, `char_in - '0'`, in the previous example. If `char_in` is a value between '0' (decimal 48) and '9' (decimal 57), then `char_in - '0'` is a value between 0 and 9.

Individual elements of an array are referenced by using subscripts. It is also possible to reference the whole array by simply using the array name without a subscript. When no subscript is specified, the array variable is treated as a pointer to the beginning of the array. For example, `digit_count` without a subscript is treated as a pointer to the beginning of the array named `digit_count`. Therefore `digit_count` and `&digit_count[0]` are equivalent expressions. A pointer to the beginning of an array is equivalent to the address of the first element in the array.

The most common array used in C is an array of characters. Even before you learned about arrays in this tutorial, you were using character arrays in the form of strings. A string, such as the format string in `printf`, is an array of characters. The word string and array of characters can be used interchangeably. There are two types of strings in C, a string variable and a string constant. A string variable is a variable declared as an array of characters (eg. `char var[8];`). A string constant is a quoted string (eg. `"abc"`).

When a string constant is used in an expression, the result is actually a pointer to the beginning of the string. For example, `printf("abc")` does not pass the string `abc` to the function `printf`. Instead it passes a pointer to where the string `abc` is stored. A string constant behaves like an array variable referenced without subscripts.

Strings and pointers are very closely related in C. You will not often use one without using the other. There are many standard library functions that require string arguments. For these functions, either a string variable or a string constant is used as an argument. In all cases, these arguments are passed as pointers. A string variable should be used if the argument is a pointer to where characters will be stored. If the characters are simply to be accessed, then either a string variable or a string constant can be used. The following example is an illustration of the relationship between string variables, string constants, and pointers. Two of the standard library functions, `strcpy` and `strcat`, are also used. `strcpy` is a function that copies one string to another. It requires two string arguments. The second argument is copied into the first. `strcat` is a function that concatenates two

strings. It also requires two string arguments. The second argument is concatenated with the first.

Example 6.4

```
                                /* Example 6.4 */
main()
{
    char name[45], first[15], middle[15], last[15];
    char *format;
    strcpy(first, "Billy ");
    strcpy(middle, "Bob ");
    strcpy(last, "Texas ");
    strcat(name, first);
    strcat(name, middle);
    strcat(name, last);
    format = "The complete name is %s\n";
    printf(format, name);
}
```

In the previous example, note that a string constant is assigned to a pointer variable, `format = "This complete name is %s\n"`, and this pointer variable is used as the first argument to `printf`. This illustrates that the use of a string constant results in a pointer to the string. Also note a new format conversion specification, `%s`. Both `scanf` and `printf` use `%s` for reading or writing strings.

A function that uses a string as an argument must have a way of detecting the end of the string. For this reason, all string constants in C have an extra character appended at the end to signal that there are no more characters in the string. This string termination character is `'\0'`, defined as `NULL` in the standard header file, `stdio`. The decimal value of this character is 0. All functions that use string arguments must check for this character to determine when the end of the string has been reached. When a variable is declared as an array of characters, you must be sure to dimension the array one character larger than the longest string it will hold. This allows a place for the `NULL` character to be stored in the array. In our previous program, `strcpy(first, "Billy ")` is used to copy the string constant `"Billy "` into the string variable `first`. The number of characters copied is actually seven, the length of the string constant plus the `NULL` character.

Back in example 3.4, you saw how you could initialize the value of a simple variable in a declaration. You can also initialize array variables in a declaration. In example 6.4 we used the `strcpy` function to assign values to the arrays `first`, `middle`, and `last`. The same results could have been achieved with the following declarations.

```

char    first[15] = "Billy ";
char    middle[15] = "Bob ";
char    last[15]  = "Texas ";

```

Each declaration allocates an array of 15 characters and assigns the characters in the quoted string to the array. For example, `first[15] = "Billy "` is equivalent to the following 7 assignment statements.

```

first[0] = 'B'; first[1] = 'i'; first[2] = 'l';
first[3] = 'l'; first[4] = 'y'; first[5] = ' ';
first[6] = '\0';

```

The size can even be omitted when initializing an array. For example, `char first[] = "Billy "` could be used. In this case, the array `first` would be allocated only seven characters of storage. This would be equivalent to using `char first[7] = "Billy "`.

Besides `scanf` and `printf`, there are two other standard functions that may be used to input or output strings. The input function is `gets` and the output function is `puts`. The function `gets` reads a string of characters from `stdin` until the newline character (`'\n'`) is encountered. `gets` replaces the newline character with the NULL (`'\0'`) string terminator character and stores the string in the array passed as an argument. The array passed to `gets` must be large enough to hold the string, allowing for the NULL character. The `puts` function writes a string of characters to `stdout`. The NULL (`'\0'`) character that terminates the string is replaced by the newline character (`'\n'`). Therefore, `gets` and `puts` are used to read and write entire lines. Take a look at this file copy example.

Example 6.5

```

#include "stdio"
#define MAXLINE 81          /* 80 characters plus NULL terminator */
main()                     /* Example 6.5 */
{
    char line[MAXLINE];

    printf("File copy from stdin to stdout\n");
    printf("Input terminates with EOF");
    while(gets(line) != NULL) puts(line);
}

```

The `gets` function returns the NULL character when the end of file is reached or an error is detected. Since the NULL character has a decimal value of 0 (false), the expression `(gets(line) != NULL)` could be replaced with simply `(gets(line))`. Note that the variable `line` is used without subscripts. What is passed to `gets` is a pointer to the beginning of the array. Using `line`

without a subscript is equivalent to `&line[0]`, the address of the first element in the array.

Arrays of more than one dimension are also allowed. The following is an example declaration for an integer array of two dimensions:

```
int bigone [5][10];
```

Individual elements of the array are referenced as follows:

```
bigone[0][0] = 10    /* first element */
bigone[1][3] = 20;   /* row 1 column 3 */
bigone[4][9] = 30;   /* last element */
```

The elements of a two dimensional array are stored sequentially by rows(row1, row2,...etc.) It can be conceptualized by the following:

```
bigone[0][0], bigone[0][1], ...bigone[0][9],
bigone[1][0], bigone[1][1], ...bigone[1][9],
bigone[2][0], bigone[2][2], ...bigone[2][9],
bigone[3][0], bigone[3][2], ...bigone[3][9],
bigone[4][0], bigone[4][2], ...bigone[4][9]
```

As noted earlier, an array name that is not followed by a subscript is treated as a pointer to the first element of the array. These pointers can be used in arithmetic expressions to index through an array, the same as using subscripts. The expression, `AA[i]`, is equivalent to the expression, `*(AA+i)`.

The result of both of these expressions is the value of the *i*th element of array `AA`.

subscript addressing		pointer addressing
<code>AA[0]</code>	! _____ !	<code>*(AA + 0)</code>
<code>AA[1]</code>	! _____ !	<code>*(AA + 1)</code>
<code>AA[2]</code>	! _____ !	<code>*(AA + 2)</code>
<code>AA[3]</code>	! _____ !	<code>*(AA + 3)</code>
<code>AA[4]</code>	! _____ !	<code>*(AA + 4)</code>

The above pointer expressions will work on any type array. Why? When a pointer variable is declared, the type specifies the size of the object. For example, `char *ptr;` declares that the variable `ptr` points to an object that is one byte in length. The declaration, `float *ptr;` declares `ptr` to point to an object that is several bytes in length, the size of a floating point number. The same applies to an array declaration, the size of each element in the array is defined.

Suppose we have a pointer to the beginning of an array of floating point

numbers. If you add one to the pointer, the result is a pointer to the second array element. The compiler takes into account the size of the object when adding one to the pointer. For example, if AA is an array of floating point numbers, and a floating point number is 4 bytes in length, then AA+1 really adds 4 to the pointer. The expression AA+2 would add 8 to the pointer, the size of two floating point numbers. Take a look at the pointer addressing in the following example.

Example 6.6

```
#define MAX 10
main()                                /* Example 6.6 */
{
    float count[MAX];
    float *ptr;
    int *iptr;
    int icount[MAX], i;

    /* initialize both arrays */
    for (i=0; i < MAX; i++) {
        count[i] = 0.3;
        icount[i] = 1;
    }

    /* assign ptr to point to sixth element of array */
    ptr = &count[5];

    *ptr = 1.;                        /* sets count[5] to 1. */
    *(ptr - 1) = .9;                  /* sets count[4] to .9 */
    *(ptr + 1) = 1.1;                 /* sets count[6] to 1.1 */

    iptr = &icount[5];
    *iptr = 0;                        /* sets count[5] to 0 */
    *(iptr - 1) = -1;                 /* sets count[4] to -1 */
    *(iptr + 1) = 2;                 /* sets count[6] to 2 */

    for (i=0; i < MAX; i++) { /* print both arrays */
        printf("count[%d] = %f  ",i,*(count+i));
        printf("icount[%d] = %d\n",i,icount[i]);
    }
}
```

The previous example shows addition and subtraction operations with pointers. These are the only arithmetic operations that are valid with pointers.

The increment and decrement operators may also be used with pointers. The increment operator applied to a pointer variable increments the variable by

the size of the object to which it points (eg. ++ptr). The decrement operator decrements the variable by the size of the object (eg. ptr--).

Two pointers can be compared using the relational operators (eg. ptr1 == ptr2). They can also be subtracted (eg. ptr1 - ptr2), but in this case both pointers must point to the exact same array. Subtracting two pointers results in the number of array elements between the two pointers.

The next example provides further illustration of the use of arrays and pointers. In fact, one of the arrays in the example is an array of pointers. That is, each element in the array is a pointer. The purpose of the program is to sort a list of names into alphabetic order. First, a list of names is read into a two dimension array of characters. In C, a two dimension array is treated as an array of one dimension, each element of which is an array. Therefore you can access a two dimension array using only a single subscript. When a single subscript is used, the result is a pointer to the beginning of the specified array element.

After the list of names is read into the two dimension array, a bubble sort algorithm is used to sort the names into alphabetic order. Another standard library function, strcmp, is used here. Strcmp takes two string arguments and compares them. It returns a negative number if the first string is less than the second, zero if the strings are identical, and a positive number if the first string is greater than the second.

Rather than sort the names within the two dimension array itself, we use another array of pointers. Each element of this array points to one of the names in the two dimension array. Then by simply sorting the pointers to the names, we can avoid having to move the names around during the sorting process. This saves time since it takes less time to move pointers than large arrays of characters. Our bubble sort algorithm is implemented by the function named sort. Its first argument is a pointer to the array of pointers and the second is the number of names to be sorted.

Example 6.7

```

/* NO_OF_NAMES is the maximum number of names */
#define NO_OF_NAMES 50
/* SIZE is the maximum number of characters in each name */
#define SIZE 31

main()
{
    int i;                /* counter */
    int number;          /* number of names read */
    char name[NO_OF_NAMES][SIZE]; /* 2 dimension array of names */
    char *nameptr[NO_OF_NAMES]; /* array of pointers to names */

```

```

/* read the names into the two dimension array */
printf("--- Enter one name per line, EOF to terminate ---\n");
for (number=0; gets(name[number]) && number<NO_OF_NAMES; number++)
    nameptr[number] = name[number];

if (number == NO_OF_NAMES)
    printf("\n *** only %d names allowed ***\n", NO_OF_NAMES);
printf("--- The names listed in alphabetic order ---\n");

/* sort the names */
sort(nameptr, number);

/* print the sorted names */
for (i=0; i<number; i++) puts(nameptr[i]);
}

```

```

sort(names, number)
char *names[]; /* array of pointers to names */
int number; /* number of names */
{
#define TRUE 1
#define FALSE 0
int notsorted = TRUE;
int i;
char *ptr;
/* sort the names by sorting the array of pointers */
/* using a bubble sort algorithm */
--number;
while (notsorted) {
    notsorted = FALSE;
    for (i=0; i<number; i++)
        if (strcmp(names[i], names[i+1]) > 0) {
            notsorted = TRUE;
            ptr = names[i]; /* swap the two pointers */
            names[i] = names[i+1];
            names[i+1] = ptr;
        }
    }
}
}

```

There are several interesting things to note about the previous program. We use a for loop to read in all the names. The loop is terminated when the function gets returns 0 (occurs when EOF is detected) or when the number of names equals the maximum that will fit in the array. Notice that name[i] is the argument passed to gets. This passes a pointer to where the ith name will be stored. After the names have been read, each element in sortedname is made to point at one of the names in the two dimension array. The sort function is then called to sort the names. Notice that the declaration of the argument names does not specify the size of the array. This is perfectly legal in C. The sort function makes multiple passes through names. For each pass, every name in the list is compared with the name following it. If the first name is greater than the second, then the names are swapped. This involves merely

swapping two pointers in names. If even a single swap is made during a pass through names, then another pass is required. When a complete pass is made without any swapping, the list of names is sorted.

Another interesting use of pointers in C is the pointer to a function. You have seen how to call a function simply by specifying its name. Now you will see how to call a function without specifying its name. Instead you call it through the use of a pointer. Although a little complicated in appearance, pointers to functions are quite useful in many applications. Using pointers to functions, a single statement in C may perform many different tasks, depending on which function is called. Now let's take a look at some declarations.

```
int f1();
int *f2();
int (*f3());
```

The first line declares a function named `f1` that returns an integer. The second line declares a function named `f2` that returns a pointer to an integer. The third line declares a variable (not a function) named `f3` that is a pointer to a function that returns an integer.

To call a function in C, you specify the function name followed by a list of arguments inside parentheses. If the function does not have any arguments, you follow the function name by an empty argument list, `()`. If you simply specify the function name without an argument list, the function is not called. Instead, a pointer to the function is the result. Looking back at our previous declarations, `f3` is a pointer variable while `f1` and `f2` are actual functions. Using an assignment statement, we can make `f3` point to either `f1` or `f2`. The statement `f3 = f1` causes `f3` to point to the function `f1`. The statement `f3 = f2` causes `f3` to point to the function `f2`.

Once `f3` has been assigned to point to a function, the function may be called by using `f3`. Consider the following example.

```
f3 = f1;
(*f3)();
```

The first line assigns `f3` the address of function `f1`. The second line calls the function `f1` through the use of the pointer.

Now let's extend our declaration of a pointer to a function to be an array of pointers to functions. The following program declares an array of 3 pointers. Each pointer is a pointer to a function that returns a double result.

Example 6.8

```

#define PI 3.141592654
main()          /* Example 6.8 */
{
    int    i;
    double sin();      /* standard library function for sine */
    double cos();     /* standard library function for cosine */
    double log();     /* standard library function for logarithm */

    /* 3 element array of pointers to functions returning double */
    double (*trig_function[3])();

    trig_function[0] = sin;
    trig_function[1] = cos;
    trig_function[2] = log;

    /* print the values of sin(PI), cos(PI), and log(PI) */
    for (i=0; i<3; i++) printf("%f\n", (*trig_function[i])(PI));
}

```

The previous program uses three standard library functions, `sin`, `cos`, and `log`. Each of these functions require one argument. Notice that they must be declared for two reasons. First, these functions return double results. All functions that return a result other than `int` must be declared. Second, we use these functions without argument lists when they are assigned to the pointer array. The compiler must know that they are functions or it will flag them as undeclared variables. Once the pointer array is assigned the three functions, it is used to execute each of the functions. The for loop prints out the value of each function using the value of `PI` as the argument.

The next example is a program that evaluates the time value of money using 4 types of calculations. Option 1 calculates what a single lump sum payment made sometime in the future is worth today. Option 2 calculates what a single lump sum payment made today would be worth at some time in the future. Option 3 calculates what the monthly installments over a period of time would be to equal a lump sum payment made today. It will also print a table showing the principle and interest paid out over the specified period of time. Option 4 calculates what a series of monthly payments made over a specified length of time is worth today. The four options are executed by calling four separate functions. These functions are called through the use of an array of pointers to functions.

Example 6.9

```
#include "stdio"

main()          /* Example 6.9 */
{
    double amount, rate, factor;
    int option, months;
    int i;

    /* declaration of functions */
    int present_value(); /* present value of future payment */
    int future_value(); /* future value of present payment */
    int monthly();      /* monthly installments on a present payment */
    int monthly_value(); /* present value of monthly installments */

    /* declaration of array of pointers to functions */
    int (*function[4])() = {present_value, future_value,
                            monthly, monthly_value};

    menu(&option,&months,&rate,&amount); /* display menu */

    for(;;) { /* loop forever */
        for (factor = 1+rate, i = 1; i < months; i++)
            factor = factor * (1 + rate);
        /* call appropriate function */
        (*function[option])(factor, amount, rate, months);
        menu(&option,&months,&rate,&amount);
    }
}
```

```

menu(option,months,rate,amount) /* display menu and collect
                                the input values */
int *option, *months;
double *rate, *amount;
{
    printf(
        "\n1) present value of a single future payment\n");
    printf(
        "2) future value of a single present payment\n");
    printf(
        "3) monthly installments on a present payment\n");
    printf(
        "4) present value of monthly installments\n");
    printf(
        "5) <<exit program>>\n\n");
    printf("    enter option --> ");
    scanf("%d",option);
    if (*option<0 || *option>4) exit(0); /* exit program */
    else (*option)--; /* make option 0..3 */
    printf("\nenter # of months --> ");
    scanf("%d",months);
    printf("enter annual interest rate (%) --> ");
    scanf("%lf",rate);
    *rate = *rate/1200;
    printf("enter dollar amount --> ");
    scanf("%lf%c",amount);
}

present_value(factor, amount)
double factor, amount;
{
    printf("the present value is %9.2f\n", amount/factor);
}

future_value(factor, amount)
double factor, amount;
{
    printf("the future value is %9.2f\n", amount*factor);
}

```

```

monthly(factor, amount, rate, months)
double factor, amount, rate;
int months;
{
    int answer, i;
    double result;
    double accum_interest = 0;
    double owed, interest, principle;

    result = factor * amount * rate / (factor-1);
    printf("the monthly payment is %9.2f\n", result);
    printf("do you wish to see the table (y or n)? ");
    answer = getchar();
    putchar('\n');
    if( answer == 'y' || answer == 'Y') {
        accum_interest = 0;
        owed = amount;
        for (i = 1; i<=months; i++) {
            interest = rate * owed;
            principle = result - interest;
            owed = owed - principle;
            accum_interest = accum_interest + interest;
            printf(" month      principle          interest  ");
            printf(" amount owed    accumulated interest\n");
            printf("  %2d      %9.2f      %9.2f",
                i,principle,interest);
            printf("          %9.2f      %9.2f\n",
                owed,accum_interest);
        }
    }
}

monthly_value(factor, amount, rate)
double factor, amount, rate;
{
    printf("the present value is %9.2f\n",
        amount * (factor-1)/(rate * factor));
}

```

There are some points to bring out about the previous program. You should notice in the main function that the elements of the array of pointers are initialized in the declaration. Any type of array may be initialized by placing values of the appropriate type inside braces {}. In this declaration, the values are pointers to functions. The values must be separated by commas. The first value is assigned to element 0, the second to element 1, and so on. Notice that the arguments to the menu function are preceded by the address of operator (&). This passes the variables by reference so that the menu function may define values for them. One of the arguments is the variable which contains the user's selected option. This option is used inside the for loop to select the appropriate function that is called via the array of pointers to functions. Notice that the for loop has three null expressions creating an infinite loop. So how does the program ever terminate. The answer to this question may be found in the menu function.

Each time through the for loop, the menu function is called to collect the input data. Option 5 of the menu is selected to terminate the program. Note that if an option less than 1 or greater than 4 is selected, the function named exit is called. This is a standard library function that causes a program to terminate. Note the last line in the menu function. The conversion specification `%*c` is used in the format string of `scanf`. The `*` is an assignment suppression symbol that causes the next value read to be discarded. The `%*c` says to read and discard the next character in the input. In this case, the next character is the newline character that is typed after entering the amount. This is actually needed only if option 3 is selected. The reason is that `scanf` skips over newline characters when reading numbers. However, option 3 requires a subsequent input of a character (the statement `answer = getchar()` in the function `monthly`). If the newline character had not been consumed by the `%*c`, then the call to `getchar` would have returned the newline character.

Another interesting thing to note about the program is that the main function calls each of the other functions with 4 arguments. If you look at the other functions, you'll see that only `monthly` requires 4 arguments. C allows you to pass fewer or more arguments to a function than the function declares. If more arguments are passed, the function simply cannot access the excess arguments since they aren't declared. If fewer arguments are passed, then the function must access only the arguments that are passed. An attempt to access a declared argument that is not passed will result in a fatal runtime error.

You should now have a fairly good understanding of how arrays and pointers are used in C programs. You've seen both one and two dimension arrays. Arrays of more than two dimensions are also allowed. You've seen various uses of pointers, including pointers to characters and pointers to functions. Up to now, all the variables used have been of the same storage class and scope. The next chapter explains what is meant by these two terms and shows you other ways of declaring variables.

Chapter 7

Storage Classes and Scoping

You have learned that all C variables have a type. They also have a storage class and what is known as scope (the part of the program over which it is defined). The variables you have seen so far are local to a specific function. They are declared within that function and no other functions have access to them. Local variables are temporary. They "come into existence" when the function is entered and "go away" when the function is exited. Their storage class is auto, meaning the local variables are automatically allocated memory when the function is entered and the memory is automatically recovered when the function terminates. A storage class may be specified just before the variable's type when it is declared, but it is not required. When no storage class is specified in the declaration of a variable, inside a function, the default storage class is auto. The following are valid declarations of auto variables:

```
main()
{
    float a;      /* defaults to storage class auto */
    auto bb;     /* defaults to type int */
    auto char cc; /* equivalent to char cc; */
}
```

Automatic variables do not retain their values between function calls and must be set every time the function is entered. If you want to make a variable available to several functions at a time, C has a storage class to define variables globally. Global variables are permanent and can be shared by two or more functions. A global variable is a variable that is declared outside a function. Because global variables are declared external to functions their storage class is called extern. The following example illustrates the declaration of a global variable:

```
int gblsave;

main()
{
}
```

All references to a global variable are to the same object. A global variable's scope is from the point in the file it is declared until the end of the file. All functions following the declaration may use the global

variable. However, if a function declares a local variable of the same name, that function cannot use the global variable. Consider the following example with the two global variables, `a1` and `b1`.

Example 7.1

```
int a1 = 1;          /* Example 7.1 */
                    /* global variable a1 */

main()
{
    a1 = 2;          /* changes the global value */
    printf("a1 inside main function = %d\n",a1);
    next();
    printf("After call to next, a1 = %d\n",a1);
    next1();
    printf("After call to next1, a1 = %d\n",a1);
}

int b1;             /* b1 is extern int */

next()
{
    char a1;        /* in next a1 is auto character */
    a1 = 'a';
    printf("a1 inside next function = %c\n",a1);
    b1 = 77;
    printf("b1 inside next function = %d\n",b1);
}

next1()
{
    float b1;       /* b1 is auto char */
    b1 = 19.3;
    printf("a1 inside next1 function = %d\n",a1);
    printf("b1 inside next1 function = %6.2f\n",b1);
    a1 = 13;
}
```

The output from the program	Class / type
al inside main function = 2	extern / int (global)
al inside next function = 'a'	auto / char (local)
bl inside next function = 77	extern / int (global)
After call to next, al = 2	extern / int (global)
al inside next1 function = 2	extern / int (global)
bl inside next1 function = 19.30	auto / float (local)
After call to next1, al = 13	extern / int (global)

Note that the global variable `al` is not accessible from the function `next` since it is declared as a local variable. Also note that the global variable `bl` is not accessible to the function `next1` for the same reason. The global variable `al` is used inside the functions `main` and `next1`, while the global variable `bl` is used only inside the function `next`.

Take a look at the changes that the variable `bl` goes through. It is globally declared between the main function and the function `next`, giving it a storage class of `extern`. Then `bl` is used inside the function `next` as a global integer. In the function `next1`, `bl` is declared to be an auto float variable, overriding the type and storage class of `bl`. If you had tried to use a variable `bl` in the main function it would be flagged as undefined by the compiler. Even though `bl` is a global variable, the C compiler doesn't know about it until the definition between `main` and `next` is read.

Globally defined variables are available for all functions in your C program. Even functions that are separately compiled may share global variables. However, you must be careful to properly declare the variables. All global variables must be defined once and only once. What you have seen previously is the definition of a global variable. Note that we called the storage class for global variables, `extern`, but did not specify this keyword in the declarations. When `extern` is not specified, the declaration is called a definition because it actually allocates storage for the variable. If `extern` is specified, no storage is allocated for the variable. It is assumed that there is a definition for the variable somewhere else, perhaps in another source file. Therefore, the keyword `extern` should be used when you wish to provide a function access to a global variable that is defined in another source file.

```
extern char stack[10];
extern int allkey, stkptr;
```

Without the keyword `extern`, the above declarations would be definitions and cause the C compiler to reserve space in memory for a global variable. With the keyword `extern` specified, the compiler does not reserve space. It assumes that the variable is defined somewhere else and merely outputs a reference to that variable. In example 7.1 you could have included an `extern` declaration for the global variable `al` in the function `next1`. The declaration is not required however, because the definition of `al` is known to the compiler when

the function `next1` is compiled. Take a look at the same example broken up into three separate compilations:

Example 7.2 - file 1

```
/* Example 7.2 - file 1 */
int al = 1;      /* global variable al */

main()
{
    al = 2;      /* changes the global value */
    printf("al inside main function = %d\n",al);
    next();
    printf("After call to next, al = %d\n",al);
    next1();
    printf("After call to next1, al = %d\n",al);
}
```

Example 7.2 - file 2

```
/* Example 7.2 - file 2 */
int bl;         /* bl is extern int */

next()
{
    char al;     /* in next al is auto character */
    al = 'a';
    printf("al inside next function = '%c'\n",al);
    bl = 77;
    printf("bl inside next function = %d\n",bl);
}
```

Example 7.2 - file 3

```

                                /* Example 7.2 - file 3 */
next1()
{
    extern int al;
    float bl;          /* bl is auto float */
    bl = 19.3;
    printf("al inside next1 function = %d\n",al);
    printf("bl inside next1 function = %6.2f\n",bl);
    al = 13;
}

```

Notice that inside the function `next1` there is a declaration for the global variable `al`. This will reference the location reserved by the definition of `al` in file 1. Also, look at the function `next`. It has a local character variable `al` that in no way conflicts with the integer global variable `al`.

You will have to learn about the linking loader to run this example. Compile each file separately, then load each file and the C standard libraries. You then can run the program. Please see the Systems Implementation Manual for details on using the linking loader.

All global variables are initialized to zero at compile time. As you can see, global variables are great for passing information between more than one function. But, they are also inherently dangerous. All functions can modify the contents of a global variable. It would be nice to allow some functions to have access to a global variable, but not all functions. The C language provides this feature in the static storage class. This class used on a global variable tells the compiler to make the name visible to all the functions in this compilation only. Thus, the variable is global to all functions in the file in which it is defined, but it hides the name from other source files. The following are static variable definitions:

```

static count; /* defaults to int */
static char name[8];

```

The next example shows the use of global static variables. The first set of functions define a data structure called a stack and the functions to manipulate the stack. Both the stack pointer, `topptr`, and the stack itself are accessible by all four functions `push`, `pop`, `top_reset`, and `stk_lst` because they are compiled together. However, these static global variables are not available to the separately compiled main function. In this manner, you build a set of functions to access and manipulate the global variables, but they are protected from being accessed by any separately compiled function. Here is the definitions of the functions, `push`, `pop`, `top_reset`, and `stk_lst`:

Example 7.3

```
/* Example 7.3 */
#define STKMAX 20 /* maximum size of the stack */

static topptr = 0; /* stack pointer */
static stack[STKMAX]; /* the stack itself */

push(vall) /* push value onto stack */
int vall;
{
    if (topptr < STKMAX)
        return( stack[topptr++] = vall);
    else {
        printf("Error - stack overflow\n");
        return(-1);
    }
}

pop ()
{
    if (topptr > 0)
        return(stack[--topptr]);
    else {
        printf("Error - stack empty\n");
        return(-1);
    }
}
```

```

top_reset()          /* reset stack */
{
    topptr = 0;
}

stk_lst()
{
    int count;

    printf("\nThe stack contains the following");
    printf(", starting at the top:\n");
    if (topptr == 0) printf("empty\n");
    else for(count = topptr - 1; count > -1; count--)
        printf("stack[%d] = %d\n",count,stack[count]);
}

```

The following is a main function that must be compiled separately. You can use it to test the stack manipulation modules just defined:

```

main()              /* Example 7.3 - file 2 */
{
    int count;      /* test push, pop, & top_reset */
    int pop(),push(),stk_lst(),top_reset();

    top_reset();   /* start off with fresh stack */
    stk_lst();     /* see if stack is empty */
    for(count=0; push(count)> -1; count++);
    stk_lst();    /* list current contents */
    while (pop() > -1);
    stk_lst();    /* see if stack empty */
}

```

The static storage class may also be used for local variables. Inside a function, the static storage class makes local variables permanent. The variables declared to be static no longer "disappear" when the function exits. This allows the variable's value to be retained over multiple function executions. The following is a short example:

Example 7.4

```

                                     /* Example 7.4 */
#define MAX 5
main()
{
    int count;

    printf("Please enter 5 numbers to be summed:\n");
    for (count = 0; count < MAX; count++)
        sumit();
    printf("Program completed\n");
}
sumit()
{
    static sum;
    int num;

    printf("Enter a number:\n");
    scanf("%d",&num);
    sum += num;
    printf("The total is %d \n",sum);
}

```

Static variables, like globals, are initialized to zero by default. However, notice that the static variable was initialized only once at compile time and not every time the function is entered.

The static storage class may also be applied to functions. When you define a function such as `sumit` in the previous example, the compiler assumes a storage class for the function of `extern`. The `extern` storage class means that the function is visible to all other functions in a program, even the functions that are compiled separately. When the `static` keyword precedes a function definition, the function acquires the static storage class. A static function is visible only to the other functions that are compiled with it.

This next example shows an example of a static function. Part numbers are entered one by one into an array, `bin`, representing a set of part bins. The array is declared to be static so that the actual details of the array implementation, such as its size, are not available to the main function. If the array has never been accessed before, then the static function `init` is called. The functions `allocbin` and `printbin` are accessible from the main function, however the function `init` is not. The use of a static function keeps you from accidentally calling the `init` function and thus destroying the part information.

Example 7.5 - file 1

```
#define TRUE 1
#define FALSE 0
#define MAX 100
static long bin[MAX];
static int bin_number = 0;
static int first = TRUE;

static init()
{
    int i;
    for(i=0; i < MAX; i++)
        bin[i] = -1;
    first = FALSE;
}

allocbin(partno)
long partno;
{
    if (first) init();
    if (bin_number < MAX)
        bin[bin_number++] = partno;
    else
        printf("Error - Out of Part Bins\n");
}

printbin()
{
    int i;
    for (i=0; i<MAX; i++)
        if (bin[i] != -1)
            printf("bin #%d = %ld\n", i, bin[i]);
}
```

Notice that the static function `init` is defined before the extern function `allocbin`. This is necessary so that the call to `init` from `allocbin` is treated as a call to a static function rather than to an extern function.

This is the main function used to enter parts into a bin.

Example 7.5 - file 2

```
main()
{
    int  allocbin(), printbin();
    long partno = 1;
    int  bin_number;
    printf("-- Enter 0 to terminate data entry --\n");
    do {
        printf("Enter part number: ");
        scanf("%ld", &partno);
        if (partno) allocbin(partno);
    } while (partno);
    printbin();
}
```

This chapter included information on the various storage classes and scoping of variables. Among the topics covered was separate compilation involving global and static variables. The next chapter discusses two features of the language that are often used together, structures and dynamic memory.

Chapter 8

Structures and Dynamic Memory

It is sometimes convenient to organize a group of related data items under a single variable name. The C language provides this ability with a data type called structure. A structure is actually a group of one or more variables that are referenced by a single name. The variables in the structure do not have to be of the same data type.

A structure is declared using the keyword `struct`, followed by a list of variable declarations enclosed by braces. Each variable in the list is considered a member of the structure. This part of the declaration may be thought of as a user defined type. Following the list of structure members is a list of variables. Each variable in this list has the type defined by the members of the structure. That is, each variable is composed of all the members of the structure. Here are some structure declarations:

```
struct {
    char f_name[8];
    char m_init[1];
    char l_name[10];
    int b_month;
    int b_day;
    int b_year;
} newperson, oldperson;

struct {
    int item_no;
    float cost;
    float retail;
} part, food_item;
```

When the above declarations are compiled, four variables are declared: `newperson`, `oldperson`, `part`, and `food_item`. The memory reserved is the amount necessary to hold all the members of the structure for each variable. For instance, the variable `food_item` is the size of one integer number plus the size of two floating point numbers.

A member of a structure variable is accessed using the C dot operator, `..`, like the following:

```

part.item_no = 999;
part.cost = .29;
newperson.fname[0] = 'A';

```

The combination of variable name, the dot operator, followed by the member name is treated just like any other simple variable. As such, it can appear anywhere in a C expression that a variable is allowed, including on the left hand side of an equal sign. The following are legal uses of member names:

```

printf(" This is the cost %d\n",part.cost);
part.retail = part.cost * 1.5;
newperson.f_name[1] = 'j';

```

The following program illustrates the use of a structure variable. The variable `input_rec` is a structure containing three members: hours, minutes, and temp.

Example 8.1

```

#include "stdio"
main()                                /* Example 8.1 */
{
    struct {
        int hours;
        int minutes;
        int temp;
    } input_rec;
    double convert(),ctemp;
    char c;

    printf("This program calculates military time");
    printf(" and centigrade temperatures\n");
    printf("Enter current time (hh:mm) : ");
    scanf("%d:%d%c",&input_rec.hours,
          &input_rec.minutes);
    printf("Is it PM ? (Y or N) ");
    c = getchar();
    if (c == 'Y' || c == 'y')
        input_rec.hours = input_rec.hours + 12;
    printf("Enter Fahrenheit temperature : ");
    scanf("%d", &input_rec.temp);
    ctemp = convert(input_rec.temp);
    printf("Time using 24-hr clock = %02d:%02d\n",
          input_rec.hours,input_rec.minutes);
    printf("Temperature is %3.1f degrees Celsius. \n",

```

```

        ctemp);
    }

double convert(ftemp)
int ftemp;
{
    return((ftemp - 32.) * 5./9.);
}

```

The variable `input_rec` is allocated enough memory to hold the three integer variables: `hours`, `minutes`, and `temp`. Notice that the address of a structure member is passed to `scanf`.

In the previous example, the variable `input_rec` was the only structure variable used and it was used only in the main function. Often when you define a particular structure, there is a need to use that structure for variables in many different functions. In such a case, you may specify a structure name that can be used when declaring variables at various places in a program.

The name appears in the declaration between the keyword `struct` and the opening brace. Take a look at the following named structures.

```

        /* structure named, no variables declared */
struct date {
    int month;
    int day;
    int year;
};

        /* structure named & pupil_in declared */
struct stu_record {
    int student_no;
    char class_id[3];
    int test[3];
    int project;
    int overall_grade;
} pupil_in;

```

The structure `date` is only a type template. The compiler does not allocate any storage for `date`, but it does keep the information about the size of the structure and the names of its members. The second declaration above, `stu_record`, allocates space for one variable by the name of `pupil_in`. Additionally, the compiler keeps the information about `stu_record`'s size and its member names for later use.

To use a previously defined structure type, you just follow the keyword `struct` by the structure's name, and then finally list any variables to be declared. These are declarations that utilize the structure definitions previously seen.

```

struct stu_record  cs_student, math_student;
struct date        birth_day, today;

```

These declarations state that `cs_student` and `math_student` are variables of type `stu_record`. Likewise, the variables `birth_day` and `today` are structures of type `date`. Memory is allocated for these variables when these declarations are processed.

Another example of the use of structure names is in the declaration of nested structures. For example, an inventory structure can contain members for the item number, the item's cost, the item's retail price, and the date it was ordered. The date itself can be a structure composed of the month, day, and year. You can even carry the nesting further by making another structure composed of the inventory information plus the date the part was shipped and its shipment number. Take a look at how you would make these declarations:

```

struct date {
    int month;
    int day;
    int year;
};

struct inventory {
    int item_no;
    float cost;
    float retail;
    struct date buy_date;
};

struct {
    struct inventory part;
    struct date ship_date;
    int shipment; /* shipment number */
} car_item;

```

To reference a shipment number you simply use `car_item.shipment`, just like you've seen before. To reference an item number on a part you would use `car_item.part.item_no`. The member-of, `.`, operator associates left to right so the compiler has no problem understanding the above construction as the `item_no` member of the structure, `part`, within the structure `car_item`. To access the month part of the shipping date you would use:

```
car_item.ship_date.month
```

However, to reference the month part of the order date you would use:

```
car_item.part.buy_date.month
```

Structure variables can also be declared as any one of the C storage classes. The scoping of these structure variables is the same as regular variables: Auto is known only to the function in which it is declared; Global static structures are known within the file they are declared; Local static structures are local to a function but retain their values permanently; Global structures are available for use by all functions in a program.

Structure variables can not be passed as arguments to functions. However, a pointer to a structure can be passed. A pointer to a structure is passed to a function using the address-of, &, operator. You must remember to declare the argument inside the function as a pointer to the structure. Consider the following example.

```

struct inventory {
    int item_no;
    float cost;
    float retail;
};

main()
{
    struct inventory newpart;
    newpart.item_no = 10;
    newpart.cost = 29.95;
    newpart.retail = 49.95;
    print_structure(&newpart);
}

print_structure(part)
struct inventory *part; /* ptr to struct */
{
    printf("no. = %d cost = %5.2f retail = %5.2f",
          (*part).item_no, (*part).cost, (*part).retail);
}

```

Notice how the structure members are accessed in the `print_structure` function. Both the `*` and `.` operators are used. Since the `.` operator has the highest precedence, the parentheses around `*part` are necessary. The expression `*part.item_no` would cause a compile error since it would be equivalent to `*(part.item_no)`, treating the member `item_no` as a pointer instead of the variable `part`.

Accessing structures using pointer variables in this way is a little messy as you can see. Luckily there is another C operator that can be used with pointers to structures. The `->` (pointer operator), which is formed by a minus sign followed by a greater than sign, will help clean up the mess. The expression `(*part).item_no` is equivalent to `part->item_no`. This looks much better doesn't it? Let's take a look at another program that uses pointers to structures.

Example 8.2

```
struct _name {
    char    last[15];
    char    first[15];
};

struct _address {
    char    street[25];
    char    city[15];
    char    state[15];
    long    zip;
};

struct label {
    struct _name name;
    struct _address address;
};

main()
{
    struct label customer;
    getlabel(&customer);
    putlabel(&customer);
}
```

```

getlabel(customer)
struct label *customer;
{
    printf("Enter Name           : ");
    scanf("%s%s%c", customer->name.first,
          customer->name.last);
    printf("Enter street         : ");
    gets(customer->address.street);
    printf("Enter city, state & zip : ");
    scanf("%s%s%ld%c", customer->address.city,
          customer->address.state,
          &customer->address.zip);
}

putlabel(customer)
struct label *customer;
{
    printf("\n%s %s\n%s\n%s %s %ld\n",
          customer->name.first,
          customer->name.last,
          customer->address.street,
          customer->address.city,
          customer->address.state,
          customer->address.zip);
}

```

Take a look at the `getlabel` function. First it uses the `scanf` function to input the first and last names. Notice that the format string is ended with `%c`. This is necessary to consume the newline character that must be typed after the last name is entered. The reason that the newline (end of line) character must be consumed is that the next input is entered using the `gets` function. Remember that the `gets` function reads until the end of line is encountered. If the newline character is not consumed by `scanf`, `gets` will think it is at the end of line and will not read any characters. The `%c` is also used to consume the newline character typed after entering the zip. This is a good practice even though it may not be necessary. Whether it is necessary or not depends on how the next input from `stdin` is read. If the next input is read using `gets` or using `scanf` with `%c`, then the newline character should be consumed. `scanf` will skip over newline characters for any format except `%c`. You should also note that the `&` (address of) operator is used only to input the long integer member, `zip`. All the other members are arrays. Remember that an array reference without subscripts is equivalent to the address of the first element of the array.

Arrays and structures can be used together so that each element of an array is a structure. A declaration might look like the following.

```

struct {
    int item_no;
    float cost;
    float retail;
} bin[5];

```

This declaration creates a 5 element array, each element of which is an entire structure. You can visualize the allocation of memory for this declaration as follows.

bin[0]	item_no	cost	retail
bin[1]	item_no	cost	retail
bin[2]	item_no	cost	retail
bin[3]	item_no	cost	retail
bin[4]	item_no	cost	retail

Each array element is a structure that can hold an `item_no`, `cost` and `retail` price. The structure members of each array element are accessed as follows.

```

bin[0].item_no  bin[0].cost  bin[0].retail
bin[1].item_no  bin[1].cost  bin[1].retail
.
.
.

```

Now let's extend example 8.2 so that more than one customer label may be entered. Since there might be multiple customer labels, they will be sorted before printing them out. For this we will use the sort function created back in example 6.7. The customer labels will be sorted using the customers last name only. Before compiling the next program, you should create three files. From example 8.2, create a file named `structs` containing the three structure definitions, `_name`, `_address`, and `label`. Create a file named `labelio` containing the `getlabel` and `putlabel` functions. Also create a file named `sort` containing the sort function from example 6.7.

Example 8.3

```

#define MAX 100
#include "stdio"      /* include standard header file */
#include "structs"    /* include the structure definitions */
#include "labelio"    /* include the getlabel and putlabel functions */
#include "sort"       /* include the sort function */

main()
{
    struct label customer[MAX];
    int    i, number = 0;
    char   more = 'Y';
    char   *names[MAX];
    while (more == 'Y' || more == 'y') {
        if (number < MAX) {
            names[number] = customer[number].name.last;
            getlabel(&customer[number++]);
            printf("More labels? (Y/N): ");
            more = getchar();
        }
        else {
            printf("* Maximum number of labels is %d *\n", MAX);
            break;
        }
    }
    sort(names, number);
    for (i=0; i<number; i++) putlabel(names[i]);
}

```

Example 8.3 declares an array of structures to hold the customer labels and an array of pointers. The array of pointers is used to point to the customer's last name in each label. This array is passed as an argument to the sort function. Notice that the elements of this array are also passed to the putlabel function. The putlabel function expects a pointer to a customer label, this being a whole structure. Due to the fact that the customer's last name is the first member in the structure, a pointer to the last name is equivalent to a pointer to the whole structure. In other words, `&customer[i]` is equivalent to `customer[i].name.last`.

Now that you are familiar with structures, let's discuss dynamic memory allocation. Dynamic memory allocation is simply the allocation of memory for a variable while the program is executing, versus allocating the memory when the program is compiled. When you define an array, the type and dimension of the array tells the compiler how much memory to allocate. When your program executes, the array will consume this amount of memory regardless of whether or not you actually use all the elements in the array. In example 8.3, we defined the size of the customer array to be 100. Even if we don't use 100

customer labels, the memory to store the 100 labels is reserved and we can't use it for anything else. This is quite wasteful since the elements of the customer array are quite large, each element being a structure containing 5 arrays and a long integer. A case such as this is where dynamic memory allocation will provide more efficient use of memory.

There are two areas of memory created for the variables in a program. The first is called the stack. This is where the compiler allocates memory for variables. Every variable that is declared is stored in the stack. The other area is called the heap. The heap is the area of memory reserved for the program's use. You control how this part of memory is allocated to variables. You can allocate memory for a variable and you can free the memory when the variable is no longer needed. There are two standard library functions, `calloc` and `cfree`, that are used just for this purpose. `Calloc` allocates the memory for a variable and `cfree` frees the memory when the variable is no longer needed.

The `calloc` function requires two arguments. The first is the number of objects for which memory will be allocated. The second is the size of each object. The `calloc` function allocates enough memory from the heap to store the objects and returns a pointer to the allocated memory. If there is not enough memory remaining to store the objects, `calloc` returns `NULL`. The return value should be checked so that you do not attempt to store objects using a `NULL` pointer. A pointer that has the `NULL` value corresponds to address 0. Storing an object at this address would most certainly be fatal to your program.

The `cfree` function requires only one argument which is a pointer to an allocated block of memory. The `cfree` function frees the block of memory at that address so that it may be reused. This function should be passed only pointers that were previously returned by the `calloc` function.

Now let's modify example 8.3 so that we use dynamic memory to store our customer labels instead of using an array. You will notice the use of another C operator in this example. The `sizeof` operator is used to get the size of a customer label. We must pass the size of a customer label to `calloc` in order to allocate the correct amount of memory. The `sizeof` operator takes one operand which may be a variable name or the name of a type. The result is the size of the variable or the type. The `sizeof` operator prevents us from having to know the size of a C data type. For example, the size of the long data type can be obtained by `sizeof(long)`. This operator allows you to write programs that do not rely on the machine dependent size of a data type. In our example, we use a structure type as the operand. The expression `sizeof(struct label)` results in the size of a customer label.

Example 8.4

```

#define MAX 100
#include "stdio"      /* include standard header file */
#include "structs"   /* include the structure definitions */
#include "labelio"   /* include the getlabel and putlabel functions */
#include "sort"      /* include the sort function */

main()
{
    int    i, number = 0;
    char   more = 'Y';
    char   *names[MAX];
    while (more == 'Y' || more == 'y') {
        if (number < MAX) {
            names[number] = calloc(1, sizeof(struct label));
            if (names[number] != NULL) getlabel(names[number++]);
            else {
                printf("<<< Out of Memory >>>\n");
                break;
            }
            printf("More labels? (Y/N): ");
            more = getchar();
        }
        else {
            printf("<<< Maximum No. of Labels is %d >>>\n", MAX);
            break;
        }
    }
    sort(names, number);
    for (i=0; i<number; i++) putlabel(names[i]);
}

```

Notice that the customer array has been eliminated. We now use only the amount of memory required to store the number of customer labels entered. If our program performed other functions, we could use the memory saved to store other variables. Note that we are still limited to a fixed number of customer labels. This limitation is imposed by the pointer array, names. However, since names is an array of pointers, each element is small. Therefore, we could make the names array larger without paying a large penalty in wasted space for the unused elements.

A data structure that can be used to eliminate the array of pointers is the linked list. A linked list, as its name implies, is a list of data items that are linked together. A structure such as our customer label could become a data item in a linked list. To turn our customer labels into a linked list, we first need a pointer variable that will point to the first label in the list. The following declaration will create a variable that we can use as a

pointer to the start of our linked list.

```
struct label *first_label;
```

Next we need a way to link the first label to the second, the second to the third, the third to the fourth, and so on. To accomplish this, we add another member (the link) to the label structure that is also a pointer to a customer label.

```
struct label {
    struct _name    name;
    struct _address address;
    struct label    *next;
}
```

Now we can link the first label to the second using the member named next. Each label in our list will use the member named next to point to the next label in the list. So now we can find the first label in the list using the first_label. Then we use the next member of the first label to find the second, and so on. We simply follow the links to get from one label to the next in our list. But how do we know when we get to the end of the list? To signal the end of our linked list, the next member of the last label in the list will have a value of NULL (0). Then all we need to do is check for this value before going to the next label.

Let's look at a program that creates a linked list for the customer labels. The program allows you to add a label, delete a label, or print all the labels in the list. The pointer to the first label in the list is first_label. First_label is declared as a global variable so that it can be used by each of the functions: add_label, delete_label, and print_labels. The function add_label adds a label to the linked list. It makes use of the function calloc to allocate memory for the label. The function delete_label deletes a label from the list. It makes use of the function cfree to free the memory used by the label. In the main function, we make use of the standard function, toupper. This function requires a single character argument and returns the character in upper case. Before compiling this program, be sure to modify the label structure (add the member next) in the file named structs created earlier.

Example 8.5

```
#include "stdio"    /* include standard header file */
#include "structs"  /* include the structure definitions */
#include "labelio"  /* include the getlabel and putlabel functions */

menu(option)
char *option;
{
    puts("\n\nA) add a label");
    puts("D) delete a label");
    puts("P) print labels");
    puts("Q) quit\n");
    printf("  Enter option --> ");
    scanf("%c%c", option);
}

struct label *first_label;

main()
{
    char option;
    for(;;) { /* loop forever */
        menu(&option);
        switch (toupper(option)) {
            case 'A' : add_label() ; break;
            case 'D' : delete_label(); break;
            case 'P' : print_labels(); break;
            case 'Q' : exit(0);
            default  : puts("*** invalid option ***");
        }
    }
}
```

```

add_label()
{
    struct label *new_label, *current_label;
    new_label = calloc(1, sizeof(struct label));
    if (new_label != NULL) {
        new_label->next = NULL;
        get_label(new_label);
        if (first_label == NULL) first_label = new_label;
        else {
            current_label = first_label;
            while (current_label->next != NULL)
                current_label = current_label->next;
            current_label->next = new_label;
        }
    }
    else printf("<<< Out of Memory >>>\n");
}

delete_label()
{
    struct label *current_label, *previous_label;
    char first[15], last[15];
    printf("Enter the name : ");
    scanf("%s%s%c", first, last);
    current_label = first_label;
    while (current_label != NULL) {
        if (strcmp(current_label->name.last, last) == 0 &&
            strcmp(current_label->name.first, first) == 0) break;
        else {
            previous_label = current_label;
            current_label = current_label->next;
        }
    }
    if (current_label == NULL) puts("*** Label not found ***");
    else {
        if (current_label == first_label)
            first_label = first_label->next;
        else
            previous_label->next = current_label->next;
        cfree(current_label);
    }
}

```

```

print_labels()
{
    struct label *current_label;
    current_label = first_label;
    puts("\n");
    while (current_label != NULL) {
        putlabel(current_label);
        current_label = current_label->next;
    }
}

```

In the `add_label` function, we create a new label and add it to the list. We must check to see if the list is empty (`first_label == NULL`) before adding the new label. If so, then the first label becomes the new label. Otherwise, we must traverse the linked list while the current label is not the last label in the list (`current_label->next != NULL`). When the current label is the last label in the list, we link it to the new label (`current_label->next = new_label`). Thus, the new label is added at the end of the linked list.

In the `delete_label` function, we prompt for the name of the label to delete from the list. Starting with the first label in the list, we compare both the first and last name entered to the name in each label. The `strcmp` function is used to compare the names. We traverse the linked list while there are still more labels in the list (`current_label != NULL`). If the label to be deleted is found, (`strcmp` returns 0), then we terminate the traversal through the list (`break`). `Current_label` will then point at the label to delete. Notice that as we traverse the list, a pointer to the previous label is maintained. We must keep this pointer because when the `current_label` is deleted, the `previous_label` must be linked to the labels following the `current_label`. Otherwise the linked list will be disconnected. Before deleting the label, we must first check to see if the label was found (`current_label == NULL`). If the label was found, then we must check to see if it is the first label in the list (`current_label == first_label`). If so, then `first_label` must be made to point to the second label in the list (`first_label = first_label->next`). Otherwise, the `previous_label` must be linked to the label that `current_label` points to (`previous_label->next = current_label->next`). Now we can delete the label (`cfree(current_label)`).

You will notice that we did not sort the list of labels. The sort function used earlier is not useful in this example since we eliminated the array of pointers that it requires as an argument. A good way to sort the labels in this example is to sort them as they are added to the list. The `add_label` function currently adds all new labels to the end of the list. We can change `add_label` so that it inserts each label into the list at its proper location. The following version of the `add_label` function sorts the labels as they are added to the list.

Example 8.6

```

add_label()
{
    struct label *new_label, *current_label, *previous_label;
    new_label = calloc(1, sizeof(struct label));
    if (new_label != NULL) {
        getlabel(new_label);
        new_label->next = NULL;
        if (first_label == NULL) first_label = new_label;
        else {
            current_label = first_label;
            while (current_label != NULL &&
                strcmp(new_label->name.last,
                    current_label->name.last) > 0) {
                previous_label = current_label;
                current_label = current_label->next; }
            if (current_label == first_label)
                first_label = new_label;
            else previous_label->next = new_label;
            new_label->next = current_label;
        }
    }
    else printf("<<< Out of Memory >>>\n");
}

```

Notice the changes that were made. We now traverse the label list while not at the end of the list (`current_label != NULL`) and the last name in the `new_label` is greater than the last name in the `current_label` (`strcmp`). Notice that we added an extra variable, `previous_label`, to point to the label preceding the current label as we traverse the list. The `new_label` will be inserted between the `current_label` and the `previous_label` when the while loop terminates. Before doing so, we must check to see if the `current_label` is the first label in the list. If so, then the `first_label` becomes the `new_label` (`first_label = new_label`). Otherwise we must point the `previous_label` to the `new_label` (`previous_label->next = new_label`). Finally, we must point the new label to the current label (`new_label->next = current_label`) to complete the link.

This chapter has attempted to give you a good exposure to the use of structures and dynamic memory allocation. You have seen arrays of structures and pointers to structures. The two standard dynamic memory functions, `calloc` and `cfree` were discussed. You were also shown how to implement a singly linked list. There are many uses for structures and dynamic memory allocation.

Chapter 9

File I/O

All the input and output up to this point has been done using the standard files, `stdin` and `stdout`. These two files, along with `stderr`, are automatically opened before a program begins execution. All three are mapped to the terminal when they are opened, `stdin` to the keyboard, `stdout` and `stderr` to the screen. We have seen three input functions that use `stdin` (`getchar`, `gets`, and `scanf`) and three output functions that use `stdout` (`putchar`, `puts`, and `printf`). The System Implementation Manual discusses how the `stdin` and `stdout` files may be remapped when a program is executed. Either may be mapped to a different device or to a disk file. The `stderr` file however, is always mapped to the screen and cannot be changed. You can do quite a lot with just these three files. But when your program needs more than one input file or more than two output files, you need the ability to create your own files.

The standard function, `fopen`, is used to open your own files. It requires two string arguments. The first argument specifies the physical name of the file. (The System Implementation Manual describes the device names that may also be used). The second argument specifies the mode in which the file is opened. The value returned by the `fopen` function is a pointer to the file. The following is the typical form of a call to the `fopen` function.

```
fp = fopen(name, mode);
```

The mode argument can have one of three values, each of which is a one character string. The possible values are "r", "w", or "a". The "r" mode opens the file for reading. The file must exist if the "r" mode is specified. Otherwise, the `fopen` function will return `NULL`. When a file is opened for reading, input starts with the first character in the file. The "w" mode opens the file for writing. The file does not have to exist if the "w" mode is specified. If the file does exist, the contents will be destroyed. When a file is opened for writing, output starts at the beginning of the file. The "a" mode opens the file for appending. This is equivalent to the "w" mode except that output starts after the last character in the file, rather than at the beginning of the file. If the file does not exist, then this mode has exactly the same effect as the "w" mode.

The only I/O functions that you've seen up to now that allow a file to be specified are `getc` and `putc`. They were used back in chapter 5 with the standard files, `stdin` and `stdout`. The `getc` and `putc` functions are the generic versions of `getchar` and `putchar`. The `getc` and `putc` functions do the same thing but are more general, allowing you to specify the input or output file.

There are also generic versions of the `scanf` and `printf` functions. The standard function, `fscanf`, is equivalent to `scanf` except that it accepts an extra argument to specify the input file. The standard function, `fprintf`, is equivalent to `printf` except that it accepts an extra argument to specify the output file. The following table shows these I/O functions with their generic equivalents.

<code>getchar()</code>	is equivalent to	<code>getc(stdin)</code>
<code>putchar(c)</code>	is equivalent to	<code>putc(c, stdout)</code>
<code>scanf(...)</code>	is equivalent to	<code>fscanf(stdin, ...)</code>
<code>printf(...)</code>	is equivalent to	<code>fprintf(stdout, ...)</code>

Now that you know about the `fopen` function, you can open your own files and perform I/O using the `getc`, `putc`, `fscanf`, or `fprintf` functions. Rather than using the standard files, `stdin` and `stdout`, you will use the file pointer returned by the `fopen` function. A variable must be declared to store the returned file pointer.

The "stdio" file defines a type named `FILE` using a feature of C that has not been discussed yet. This is the `typedef`, which stands for type definition. A type definition simply defines a name that can be used to declare variables. This name can then be used just like any of the predefined data types such as `char`, `int`, `float`, etc. So in effect, `typedef` allows you to define your own data types. To define a type, you use the keyword `typedef`, followed by a predefined data type or your own defined data type, followed by the name by which the data type will be referenced. The following are examples of type definitions.

```
typedef char  CHARACTER;
typedef int   INTEGER;
typedef float REAL;
typedef struct {
    char last[15];
    char first[15];
} NAME;
```

User defined data types are typically given upper case names to distinguish them from the predefined data types. We can now use the above type definitions to declare variables.

```
CHARACTER c;      /* equivalent to char c;      */
INTEGER   i;      /* equivalent to int  i;      */
REAL      r;      /* equivalent to float r;     */
NAME      name;   /* equivalent to struct {
                                char last[15];
                                char first[15];
                            } name;   */
```

The type definition named `FILE` in the "stdio" file is used to declare pointers to files. Let's look at an example program that copies one file to

another using `getc` and `putc` and our own file variables.

Example 9.1

```
#include "stdio"

main()
{
    FILE *input_file, *output_file;
    char  input_name[15], output_name[15];
    int   c;

    puts("*** File Copy Program ***");

    printf("Enter the name of the input file : ");
    scanf("%s", input_name);
    printf("Enter the name of the output file : ");
    scanf("%s", output_name);

    input_file = fopen(input_name, "r");
    if (input_file == NULL) {
        puts("*** Can't open input file ***");
        exit(0);
    }

    output_file = fopen(output_name, "w");
    if (output_file == NULL) {
        puts("*** Can't open output file ***");
        exit(0);
    }

    while ((c = getc(input_file)) != EOF)
        putc(c, output_file);
}
```

Notice that we open the input file using mode "r" and the output file using mode "w". We could have used mode "a" for the output file if we wished the input file to be appended to the end of an existing file.

There are two standard functions that are slight variations of the `gets` and `puts` functions. These are naturally called `fgets` and `fputs` since both have a file pointer as one of the arguments. Both `gets` and `fgets` read a string until the end of line (newline character '\n') is encountered. The difference between the two functions is that `fgets` includes the newline character as part of the string while `gets` does not. `Fgets` also has an argument to specify the maximum number of characters that will be read if an end of line is not encountered. Both `puts` and `fputs` output a string of characters until the NULL character is encountered. The difference between `puts` and `fputs` is that `puts`

outputs a newline character at the end of the string while fputs does not. Both gets and fgets return NULL when the end of file is detected or an error occurs during input while puts and fputs return EOF if an error occurs during output.

With the exception of the handling of the newline character, the following function calls are equivalent. Infinity is used to represent an infinite number.

```

    gets(s)    is equivalent to    fgets(s, infinity, stdin)
    puts(s)    is equivalent to    fputs(s, stdout)

```

Let's modify example 9.1 so that it uses fgets and fputs to perform the file copy. All the other functions used in example 9.1 will be converted to their generic counter parts.

Example 9.2

```

#include "stdio"
#define BUFSIZE 81
main()
{
    FILE *input_file, *output_file;
    char input_name[15], output_name[15], buffer[BUFSIZE];
    int c;
    fputs("*** File Copy Program ***\n", stdout);

    fprintf(stdout, "Enter the name of the input file : ");
    fscanf(stdin, "%s", input_name);
    fprintf(stdout, "Enter the name of the output file : ");
    fscanf(stdin, "%s", output_name);

    input_file = fopen(input_name, "r");
    if (input_file == NULL) {
        fputs("*** Can't open input file ***", stdout);
        exit(0);
    }
    output_file = fopen(output_name, "w");
    if (output_file == NULL) {
        fputs("*** Can't open output file ***", stdout);
        exit(0);
    }
    while (fgets(buffer, BUFSIZE, input_file) != NULL)
        fputs(buffer, output_file);
}

```

Now let's modify example 8.4 to make it a little more versatile. Example

8.4 is a program that reads labels from the keyboard and then sorts and prints the labels on the terminal screen. Our next example will modify this program so that the labels may optionally be read from a file and the sorted labels written to a file. The only part of example 8.4 that will remain unchanged is the sort function. You will notice the use of another standard C function, `strlen`. The `strlen` function takes a single string argument and returns the length of the string.

Example 9.3

```
#define MAX    100
#include "stdio"    /* include standard header file */
#include "sort"    /* include the sort function */

typedef struct {
    char    last[15];
    char    first[15];
} NAME;

typedef struct {
    char    street[25];
    char    city[15];
    char    state[15];
    long    zip;
} ADDRESS;

typedef struct {
    NAME    name;
    ADDRESS address;
} LABEL;

getlabel(fp, customer)
FILE *fp;
LABEL *customer;
{
    if (fp == stdin) printf("Enter Name          : ");
    fscanf(fp, "%s%s%c", customer->name.first,
           customer->name.last);
    if (fp == stdin) printf("Enter street          : ");
    fgets(customer->address.street, 25, fp);
    if (fp == stdin) printf("Enter city, state & zip : ");
    return fscanf(fp, "%s%s%ld%c", customer->address.city,
                 customer->address.state,
                 &customer->address.zip);
}
```

```
putlabel(fp, customer)
FILE *fp;
LABEL *customer;
{
    fprintf(fp, "\n%s %s\n%s%s %s %ld\n",
            customer->name.first,
            customer->name.last,
            customer->address.street,
            customer->address.city,
            customer->address.state,
            customer->address.zip);
}
```

```

main()
{
    int    i, number = 0;
    char   more = 'Y', *names[MAX], in_name[15], out_name[15];
    FILE   *in_file, *out_file;
    puts("*** Label Sorting Program ***");
    puts("press the enter key to map files to terminal\n");
    printf("Input file containing unsorted labels : ");
    gets(in_name);
    printf("Output file to contain sorted labels : ");
    gets(out_name);
    if (strlen(in_name) == 0) in_file = stdin;
    else {
        in_file = fopen(in_name, "r");
        if (in_file == NULL) {
            printf("Can't open input: %s", in_name); exit(0);
        }
    }
    if (strlen(out_name) == 0) out_file = stdout;
    else {
        out_file = fopen(out_name, "w");
        if (out_file == NULL) {
            printf("Can't open output: %s", out_name); exit(0);
        }
    }
    while (more == 'Y' || more == 'y') {
        if (number < MAX) {
            names[number] = calloc(1, sizeof(LABEL));
            if (names[number] != NULL) {
                if (getlabel(in_file, names[number]) == EOF) {
                    cfree(names[number]); break;
                }
                ++number;
                if (in_file == stdin) {
                    printf("More labels? (Y/N): ");
                    more = getchar();
                }
            }
        }
        else {
            printf("<<< Out of Memory >>>\n");
            break;
        }
    }
    else {
        printf("<<< Maximum No. of Labels is %d >>>\n", MAX);
        break;
    }
}
sort(names, number);
for (i=0; i<number; i++) putlabel(out_file, names[i]);
}

```

First of all, you should notice that we changed all the structure

definitions to use typedef. Next we added an extra argument to both the `getlabel` and `putlabel` functions, a file pointer.

Inside `getlabel`, we check to see if the input is coming from `stdin` (the keyboard). If so, then we provide input prompts for the user. If the input is coming from a file, these prompts are unnecessary. Also notice the addition of the `return` statement with the last `scanf` function call. Now that the `getlabel` function can get its input from a file, we must detect when the end of file has been reached. The `scanf` function normally returns the number of items successfully input each time it is called. However, if the end of file is detected before all the input is complete, `scanf` returns EOF. The `return` statement specifies that the value returned by `scanf` will also be returned by the `getlabel` function.

Inside `putlabel`, we eliminated a newline character in the format string. Since the `getlabel` function uses `fgets` to input the street, the street array already has a newline character at the end of the string.

In the main function, the first order of business is to prompt the user for the input and output file names. We allow the user to specify input from the keyboard by merely pressing the enter (return) key. The screen may be specified for the output by merely pressing the enter key as well. Otherwise, the user may type in a valid file or device name for either the input or output file. We then use the `strlen` function to determine whether or not the user entered a file name or merely pressed the enter key. If the `in_name` or `out_name` has a length of 0, then the user merely pressed the enter key. In this case we map the input to the keyboard (`in_file = stdin`) and the output to the screen (`out_file = stdout`). Otherwise, we use the `fopen` function to open the files with the user supplied names. Note that we check to see that the files are successfully opened. If the files are not successfully opened, we display an appropriate message and exit the program.

The input file is used to collect the labels. If the input is coming from a file, the input process is terminated when the end of file is reached. Notice that on each call to the `getlabel` function, the returned value is compared to EOF. When EOF is returned by `getlabel`, we free the memory allocated to the last label (`cfree(names[number])`) since it is not used and terminate the loop. If the input is coming from the keyboard, we prompt the user after each label is entered. When the user answers this prompt with a character other than 'Y' or 'y', the loop is terminated. Therefore, we have two methods of terminating the while loop, depending on where the input is coming from.

For our last example, we will illustrate the use of the append mode with `fopen`. We will also introduce the standard function for closing files, `fclose`. The append function in our program accepts two file names as arguments, opens the two files, and then appends the first file to the second. Notice that these two arguments are declared as pointers to `char`. They could have been declared as arrays, but since we do not need to access the individual array elements, pointer to `char` works equally as well. At the end of the append function, the two open files are closed using the `fclose` function. The `fclose` function requires a single file pointer as an argument and closes the file to which it points. All open files are closed

automatically when a program terminates. However, `fclose` provides a means of explicitly closing files. It is a good practice to close a file when it is no longer needed. If a program abnormally terminates, any output files that are open may be lost. There is also a limit to the number of files you can have open at any one time. This limit is defined by the `MAXFILES` constant in the `"stdio"` file. By using `fclose`, you may free a file pointer for use with another file.

Example 9.4

```
#include "stdio"

main()
{
    char input_name[15], output_name[15];

    puts("*** Append File Example ***");
    fputs("Enter the input file name : ", stdout);
    gets(input_name);
    fputs("Enter the output file name: ", stdout);
    gets(output_name);
    if (append(input_name, output_name) == NULL)
        puts("Dismal Failure");
    else
        puts("Extremely Successful");
}
```

```
append(name1, name2)
char *name1, *name2;
{
    #define FAILURE 0
    #define SUCCESS 1
    FILE *input, *output;
    int c;

    input = fopen(name1, "r");
    output = fopen(name2, "a");
    if (input == NULL) {
        fprintf(stderr, "Can't open input: %s\n", name1);
        return FAILURE;
    }
    if (output == NULL) {
        fprintf(stderr, "Can't open output: %s\n", name2);
        return FAILURE;
    }
    while ((c = getc(input)) != EOF) putc(c, output);
    fclose(input);
    fclose(output);
    return SUCCESS;
}
```

Well, this pretty well does it for the tutorial. I hope that you have learned enough to start feeling comfortable with the language. The only way to really feel comfortable is to write a lot of programs. The more you write, the easier it becomes. The C language provides an infinite number of ways to accomplish your programming tasks and this tutorial has presented only a few of them. The Reference Manual describes many more features of the language that were not even mentioned here. There are also a large number of functions that were not discussed. The Reference and System Implementation Manuals describe all of the functions provided with this implementation of C. The System Implementation Manual describes those functions which are machine dependent while the Reference Manual describes those that are machine independent.

Index

!, not operator 36
!=, not equal operator 29
" ", string delineator 5
#define 19
#include 51
%, modulo operator 11
%d, integer format 10
%e, conversion specification 49
%f, floating format 13
%x, conversion specification 49, 64
&&, and operator 36
&, address-of operator 17
* operator 63
* suppression character 37
*, multiplication operator 11
*/, ending comment 6
++, increment operator 25
+, addition operator 11
, comma operator 38
-, subtraction operator 11
-, unary minus 8
--, decrement operator 25
/*, beginning comment 6
/, division operator 11
;, statement terminator 4
<, less than operator 29
<=, less than operator 29
=, assignment operator 9
== vs = 35
==, equality operator 29
>, greater than operator 29
>=, greater than operator 29
?:, conditional expression operator 57
\x, escape sequence 6
appending 107
arithmetic operators 11
array 65
array elements 65, 65
array of characters 67
array of pointers 72
array, base of 66
assignment operator 25
associativity 21
auto storage class 81
block 4
break statement 55
call by value 17, 45
calloc 100

case label 58
cfree 100
char data type 11
char declarations 11
character constant 11
character format 13
comment 6
compound statement 32
conditional statements nested 39
constants 7
contents-of operator 63
continue statement 57
control statement 29
control statement nesting 33
cos 75
data types 8
declarations, integer variables 8
default label 58
do statement 39
dynamic memory 99
else statement 31
EOF 52
escape sequence 6
example 1.2 4
example 1.3 5
example 1.4 9
example 1.5 13
example 2.1 16
example 2.2 18
example 2.3 20
example 2.4 22
example 2.5 23
example 2.6 26
example 3.1 29
example 3.2 31
example 3.3 32
example 3.4 33
example 3.5 35
example 3.6 37
example 3.7 38
example 3.8 39
example 3.9 40
example 4.1 41
example 4.2 45
example 4.3 47
example 4.4 48
example 5.1 51
example 5.2 53
example 5.3 54
example 5.4a 55
example 5.4b 56
example 5.5 57
example 5.6 58
example 5.7 59
example 5.8 60
example 6.1 64

example 6.2 65
 example 6.3 66
 example 6.4 68
 example 6.5 69
 example 6.6 71
 example 6.7 72
 example 6.8 75
 example 6.9 76
 example 7.1 82
 example 7.2 84
 example 7.3 86
 example 7.4 88
 example 7.5 - file 1 89
 example 7.5 - file 2 90
 example 8.1 92
 example 8.2 96
 example 8.3 99
 example 8.4 101
 example 8.5 103
 example 8.6 106
 example 9.1 109
 example 9.2 110
 example 9.3 111
 example 9.4 115
 exit function 79
 extern storage class 81
 fclose 114
 fgets 109
 floating point data type 11
 flow of control 29, 41
 fopen 107
 for statement 37
 fprintf 108
 fputs 109
 fscanf 108
 function arguments 17, 43
 function call 4, 17
 function call in expressions 42
 function declaration 47
 getc 107
 getc 52
 getchar 54
 gets 69, 69
 global variables 81
 goto statement 56
 heap 100
 if statement 30
 include files 51
 indirection operator 63
 initialization, array 78
 integer constant 8
 log 75
 logical operators 36
 long integer 12
 member of operator, . 92
 member of, structure 92

nested structures 94
null statement 34
operator precedence 21
pointer addressing 70
pointer data type 63
pointer initialization 63
pointer to function 74
pointers 63
postfix operator 25
prefix operator 25
printf 6
printf, arguments 10
putc 107
putc 52
putchar 54
puts 69
reading 107
relational operators 29
return statement 44
scanf 16
scope of global variable 81
scoping 81
separate compilation 83
sin 75
sizeof operator 100
stack 100
static function 88
static storage class 85
static, internal 87
stderr 53
stdin 15, 53
stdout 15, 53
storage class 81
storage class, extern 81
storage class, static 85
strcat 67
strcpy 67
string 67
structure declaration 91
structure pointer operator, -> 95
structure, pointer to 95
structures 91
structure names 93
switch statement 58
symbolic constant 19
toupper 102
truncation 22
type conversion, automatic 22
typedef 108
unsigned integer 12
variables 7
void 47
while statement 32
whitespace 7
writing 107
||, or operator 36

Table of Contents

Chapter 1 Program Elements	1
1.1 Identifier	1
1.2 Constants	2
1.2.1 Integer Constants	2
1.2.2 Floating Point Constants	3
1.2.3 Character Constants	3
1.2.4 Symbolic Constants	4
1.2.5 String Constants	5
1.3 Reserved Words	6
1.4 Operators	7
1.5 Alternate Symbols	7
1.6 Comment	8
1.7 Semicolon	8
1.8 Brace	9
1.9 Terminology	10
Chapter 2 Program Structure	11
2.1 Functions	11
2.1.1 Function Header	13
2.1.2 Argument Declarations	13
2.1.3 Function Body	14
2.2 Local Variables	15
2.3 Global Variables	16
Chapter 3 Basic Data Types and Declarations	19
3.1 Character Variables	19
3.2 Integer Variables	20
3.2.1 Short Integers	20
3.2.2 Long Integers	21
3.2.3 Unsigned Integers	22
3.3 Floating Point Variables	22
3.3.1 Double Precision	23
3.4 Storage Classes	24
3.4.1 Auto Variables	24
3.4.2 Extern Variables	25
3.4.3 Static Variables	27

3.4.4 Register Variables	28
3.5 Initialization of Basic Data Types	29
3.6 Type Definitions	29
Chapter 4 Basic Operators and Expressions	31
4.1 Operator Precedence and Grouping	31
4.2 Assignment Operator	34
4.3 Arithmetic Operators	34
4.3.1 Properties of + Operator	35
4.3.2 Properties of - Operator	35
4.3.3 Properties of * Operator	36
4.3.4 Properties of / Operator	36
4.3.5 Properties of % Operator	37
4.4 Relational and Equality Operators	37
4.4.1 Properties of < Operator	39
4.4.2 Properties of <= Operator	39
4.4.3 Properties of > Operator	39
4.4.4 Properties of >= Operator	39
4.4.5 Properties of == Operator	39
4.4.6 Properties of != Operator	39
4.5 Logical Operators	40
4.5.1 Properties of && Operator	40
4.5.2 Properties of Operator	41
4.5.3 Properties of ! Operator	41
4.6 Type Conversions	42
Chapter 5 More Operators and Expressions	43
5.1 Increment and Decrement Operators	43
5.1.1 Properties of ++ Operator	44
5.1.2 Properties of -- Operator	44
5.2 Bitwise Operators	46
5.2.1 Properties of ~ Operator	46
5.2.2 Properties of >> Operator	47
5.2.3 Properties of << Operator	48
5.2.4 Properties of & Operator	48
5.2.5 Properties of ^ Operator	49
5.2.6 Properties of Operator	50
5.3 Assignment Operators	51
5.4 Address Of and Contents Of Operators	53
5.4.1 Properties of & Operator	53
5.4.2 Properties of * Operator	54

5.5	Sizeof Operator	54
5.6	Cast Operator	55
5.7	Comma Operator	56
5.8	Structure Member Operator	56
5.9	Structure Pointer Operator	57
5.10	Conditional Expression	58
5.11	Constant Expressions	59
5.12	Sample Expressions	59
Chapter 6 Functions		61
6.1	Function Definition	61
6.1.1	Function Names	62
6.1.2	Function Types	62
6.1.3	Function Arguments	63
6.1.4	Function Body	63
6.2	Nested Blocks	65
6.3	The Main Function	65
6.4	Static Functions	66
6.5	Calling Functions	67
6.6	Function Pointers	67
6.7	Recursion	68
Chapter 7 Pointers and Arrays		71
7.1	Pointers	71
7.2	Arrays	72
7.3	Using Pointers with Arrays	74
7.4	Array Initialization	75
Chapter 8 Structures and Unions		77
8.1	Structures	77
8.1.1	Defining Structures	77
8.1.2	Referencing Structures	80
8.1.3	Structure Initialization	80
8.1.4	Pointers to Structures	81
8.1.5	Arrays of Structures	82
8.1.6	Bit Fields	83
8.2	Unions	84

Chapter 9 Statements	87
9.1 Simple and Compound Statements	87
9.2 Conditional Statements	87
9.2.1 if	87
9.2.2 else	88
9.2.3 switch	89
9.3 Looping Statements	90
9.3.1 while	90
9.3.2 do-while	91
9.3.3 for	92
9.4 break	92
9.5 continue	93
9.6 goto and labels	94
9.7 return	95
9.8 null	95
Chapter 10 Input and Output	97
10.1 Opening and Closing Files	97
10.1.1 fopen	98
10.1.2 fclose	99
10.2 Character I/O	100
10.2.1 getchar	100
10.2.2 putchar	101
10.2.3 getc	101
10.2.4 putc	102
10.2.5 ungetc	103
10.3 String I/O	104
10.3.1 gets	104
10.3.2 puts	105
10.3.3 fgets	105
10.3.4 fputs	106
10.4 Formatted I/O	107
10.4.1 Input Format Strings	108
10.4.2 Output Format Strings	111
10.4.3 scanf	115

10.4.4	printf	116
10.4.5	fscanf	117
10.4.6	fprintf	118
10.4.7	sscanf	119
10.4.8	sprintf	121
Chapter 11 Standard Functions		123
11.1 Character Functions		123
11.1.1	isalpha	123
11.1.2	isdigit	124
11.1.3	isspace	125
11.1.4	islower	125
11.1.5	isupper	126
11.1.6	tolower	127
11.1.7	toupper	128
11.2 String Functions		128
11.2.1	strlen	129
11.2.2	strcpy	129
11.2.3	strcmp	130
11.2.4	strcat	131
11.2.5	strsave	132
11.3 Dynamic String Functions		133
11.3.1	stods	133
11.3.2	dstos	134
11.4 Conversion Functions		135
11.4.1	atoi	135
11.4.2	atof	136
11.4.3	itoa	137
11.4.4	ftoa	137
11.5 Dynamic Memory Functions		139
11.5.1	calloc	139
11.5.2	cfree	141
11.6 Math Functions		142
11.6.1	abs	142
11.6.2	atan	142
11.6.3	cos	143
11.6.4	exp	144
11.6.5	log	144
11.6.6	sin	145
11.6.7	sqr	145
11.6.8	sqrt	146
11.7 Termination Functions		147

11.7.1	exit	147
11.7.2	_exit	148
Chapter 12 Compiler Controls		149
12.1	Preprocessor Statements	149
12.1.1	#include	149
12.1.2	#define	150
12.1.3	#undef	151
12.1.4	#ifdef	151
12.1.5	#ifndef	152
12.1.6	#if	153
12.1.7	#else	154
12.1.8	#line	155
12.2	Compiler Options	156
12.2.1	CONVERT Option	156
12.2.2	LIST Option	157
12.2.3	LISTMACRO Option	158
12.2.4	NESTCMNT Option	158
12.2.5	PAGESIZE Option	159
12.2.6	SIGNEXT Option	159
12.2.7	UPPERCASE Option	160
12.2.8	WIDELIST Option	161
12.2.9	ZERO Option	162
Appendix A Error Messages		163
A.1	Compiler Error Messages	163
A.2	Runtime Error Messages	165
Appendix B ASCII Table		169
Appendix C Differences from Kernighan and Ritchie		173

Chapter 1

Program Elements

1.1 Identifier

An identifier serves to denote the program name, a constant, a type, a variable, or a function. An identifier is made up of a sequence of letters and digits. It has the following characteristics:

1. The first character must be a letter.
2. The underscore (`_`) and dollar sign (`$`) count as letters.
3. Upper and lower case letters are different (i.e. Letter does not equal LETTER, which does not equal letter, etc.).

The length of an identifier is arbitrary but only the first eight (8) characters are significant to the compiler. For example, `student_name` and `student_grade` would be taken to be identical because the compiler discards all characters past the eighth character. The identifier cannot contain blanks or span (cross) a line boundary.

By default, the compiler will generate upper case function names in the object code even if lower case is used in the source file. The compiler option `/*$NO UPPER CASE*/` may be used to allow lower case function names.

Examples:

<code>student-name</code>	incorrect, hyphen (-) not allowed
<code>Net_profit</code>	correct
<code>square root</code>	incorrect, embedded blank
<code>I</code>	correct
<code>7_card_stud</code>	incorrect, begins with a digit
<code>seven_card_stud</code>	correct
<code>SQUAREROOT</code>	correct
<code>a2653</code>	correct
<code>July_28_1982</code>	correct

1.2 Constants

Constants can be integer, floating point, character, or string.

1.2.1 Integer Constants

An integer constant is a sequence of digits. Normally, integer constants are specified in decimal (base 10). An integer may be specified in octal (base 8) by starting the digit sequence with a 0. Octal is often used when the integer is representing a specific bit pattern. An integer may be specified in hexadecimal (base 16) by preceding the number with 0x or 0X. Hexadecimal is typically used when the integer is representing a memory address.

Integer constants may be allocated storage of type int, unsigned int, or long int, depending on the size and/or sign of the constant. Storage is allocated based on the following rules.

1. If the constant is in the range of type int, then it is allocated storage of type int.
2. Else the constant is allocated storage of type long.

See the System Implementation Manual for the ranges of these types.

An integer constant may be forced to type long by appending l (the letter ell) or L to the digit sequence.

Examples:

58	correct, decimal
072	correct, octal
0x5a	correct, hexadecimal
FACE	incorrect, does not begin with 0x or 0X
092	incorrect, octal numbers cannot have a 9
-265	correct, decimal
1,223	incorrect, cannot have a comma in an integer
423L	correct, long decimal
0x9A7EL	correct, long hexadecimal
086L	correct, long octal

1.2.2 Floating Point Constants

A floating point constant may contain several parts: a whole part, a decimal point, a fractional part, an e or E, and an exponent. The whole part, fractional part, and exponent are digit sequences. The whole part and the exponent may optionally be preceded by a + or - sign. All floating point constants are taken to be double precision.

There are times when using all the parts of the floating point constant is cumbersome. C allows some shorthand notations. For example, we could write 3.102e+3 as a floating point constant. However, C also accepts 3102. or 3.102e3 or 0.3102E4 to produce the same value for the floating point constant. Floating point constants do have some restrictions on format.

1. Either the whole or the fractional part may be missing, but not both.
2. Either the decimal point or the E or e may be missing, but not both.

Examples:

-638.	correct
0.638e-2	correct
638	incorrect, both . and e (or E) are missing
638e-7	correct
E4	incorrect, both integer and fractional part missing
245.1E6	correct
2,451E+6	incorrect, comma not allowed in floating point
638E7.	incorrect, exponent not an integer

1.2.3 Character Constants

A character constant is a character enclosed in single quotes ('). For instance, 't', 'K', and 'b' are all examples of character constants. The value of the character constant is the numerical value of the character in ASCII (see table in the Appendix). The character constant 't' has the numerical value of 116 (decimal), 164 (octal), or 74 (hexadecimal). Certain non-graphic characters which are not printable may be represented by escape sequences as shown in the following table.

Name -----	Escape Sequence -----
newline (NL)	\n or \N
horizontal tab (HT)	\t or \T
backspace (BS)	\b or \B
carriage return (CR)	\r or \R
line feed (LF)	\l or \L
form feed (FF)	\f or \F
backslash (\)	\\
single quote (')	\'
double quote (")	\"
null character (NUL)	\0
bit pattern	\ddd

The escape `\ddd` is a backslash (`\`) followed by 1 to 3 octal digits which represent the value of the character constant. For example, the `'t'` from above could also be expressed as `'\164'`. The null character is a special case and represents the character whose value is 0 (zero). If the character which follows the `\` is not one of those in the table, the `\` is ignored. In all cases, the escape sequence is taken as one character even though 2 or more characters may be used to represent it.

Examples:

```
't'      correct
'\122'   correct (this is the same as 'R')
"b"      incorrect, must be enclosed in single quotes
'the'    incorrect, must be a single character
'Q'      correct
'\n'     correct (this is the newline character sequence)
'\0'     correct (this is the NULL character)
```

1.2.4 Symbolic Constants

A symbolic constant is used to give a descriptive name to a program constant. The name is then used throughout the program in place of the constant itself. This makes it easy to change the value of the constant at some latter time by confining its occurrence to one place in the program. The preprocessor macro, `#define`, is used to define a symbolic constant. This macro takes two strings as its parameters. The first string is the symbolic name for the constant and the second is the actual text to be used wherever the symbolic name appears in the program. The compiler replaces all unquoted occurrences of the symbolic name by the corresponding defined replacement text.

Suppose we wanted to define two constants where 1 is on and 0 is off. The following might be used.

```
#define ON 1
#define OFF 0
```

This defines symbolic names for the constants 1 and 0. Notice there is no semicolon (;) at the end of the #define statement. This is because the semicolon would be taken as part of the replacement text. Suppose the following constant was defined.

```
#define SIX 6;
```

Then each occurrence of SIX would be replaced by 6; which is probably not what was intended. Note that the symbolic constant is written in uppercase letters to make it is easy to distinguish from other program identifiers. Also note that the replacement text is not limited to numbers. Any string of characters may be used.

Examples:

```
#define MAX_COLUMNS 80
#define MAX_ROWS 24
#define NAME Mary Jane
#define COST 9.95
```

1.2.5 String Constants

A string constant is a sequence of zero (0) or more characters enclosed in double quotes ("). A string constant is of the same type as an array of characters. The double quotes (") serve as delimiters and are not part of the string itself. The compiler places a null byte (\0) at the end of every string to mark the end of the string. If a double quote (") is part of the string, it must be preceded by a backslash (\). The escape sequences shown for character constants may be used in a string since the escape sequences represent characters. A string with 0 (zero) characters is said to be a null string and is represented as "".

Examples:

```

"This is a string."      correct
"This is a "string"."   incorrect, double quotes surrounding
                        string are not preceded by \
"This is a \"string\"." correct
"\103\122\124"         correct (same as "CRT")
"I am a 'C' wizard."    correct
"I am a 'C' wizard. \n" correct
""                       correct (null string)
" "                      correct, string containing one
                        blank character (not the same as ' ')

```

A string constant may span more than one line by terminating the string with the (\) character.

Example:

```

"This is a string constant that spans\
more than one line."

```

1.3 Reserved Words

The following words have special meaning to the compiler and are reserved. They cannot be used as identifiers. The reserved words may be entered in upper or lower case. Since these words may be in any case, then AUTO, Auto, and auto are all taken to be reserved words.

auto	double	if	static
break	else	int	struct
case	entry	long	switch
char	extern	register	typedef
continue	float	return	union
default	for	short	unsigned
do	goto	sizeof	void
while			

1.4 Operators

Operators in C consist of a set of predefined symbols. These predefined symbols vary in length from one to three characters. The C compiler will match the longest possible symbol. For example, `b--2` will be matched as `b--` and `2`, and not as `b - (-2)`.

```

+   -   *   /   %   &   ^   |
<   >   <=  >=  <<  >>  <<= >>=
{   }   (   )   [   ]   /*  */
.   ->  ;   ,   :   ?   !   ~
=   +=  -=  *=  /=  %=  ==  !=
&=  ^=  |=  ++  --  \   &&  ||

```

1.5 Alternate Symbols

Since some keyboards may not contain all the special symbols used for C operators, a set of alternate symbols has been defined. These alternate symbols may be substituted for the normal symbols if necessary.

C Symbol	Alternates
~	!!
[(@
]	@)
{	(#
}	#)
	/!
\	//
	/!!
^	@

1.6 Comment

Comments are used to document the code used in a program. A comment is a sequence of characters beginning with `/*` and ending with `*/`. All characters that fall between these two symbols are ignored by the compiler.

Comments may begin and end anywhere on a line as long as they don't enclose part of the program itself. Comments may also cross line boundaries. Normally, comments may not be nested. That is, a comment may not appear inside another comment. However, there is a compiler option that can be turned on to allow nested comments.

Examples:

```
/* fed and state income taxes */
/* withheld from this paycheck */
--correct

/* retirement contribution */
--correct

/* One small step for C, one giant leap for me. */
--correct

/* I am an old C dog because I write /*comments*/. */
--incorrect unless compiler option for nesting is used

/* federal and state income taxes
   withheld from this paycheck */
--correct because comments can cross line boundaries
```

1.7 Semicolon

The semicolon (`;`) is used by C as a statement terminator, not as a statement separator. Any expression followed by a `;` becomes a statement.

1.8 Brace

The braces { and } are used in C to group declarations and statements together into blocks or (compound statements). This makes the compound statements equivalent to a single statement, much the same as the begin and end in Pascal. Because braces are not statements, semicolons are never placed after a right brace (}). C allows declarations to be made in any block immediately after the left brace ({).

Example:

```
if (x < y) {
    x += y;    /* this is a compound statement */
    printf("%d %d \n", x, y);
}
else {
    y += x;    /* this is a compound statement */
    printf("%d %d \n", x, y);
}
```

1.9 Terminology

The following is a list of terms which will be used in the remainder of the reference manual.

whitespace	A blank, tab, or newline character.
expression	An expression consists of an identifier or a sequence of identifiers and operators.
object	An area of memory where data may be manipulated.
scalar	A simple (single) variable. That is, a variable that is of type int, float, double, long, short, unsigned, char, or pointer to one of those types (see Chapter 3 for more information on data types). A scalar is not a composite of several variables. Arrays and structures, for example, are not scalars.
lvalue	An identifier or expression which may appear on the left of an assignment operator (left of an equals sign). An lvalue is an expression which refers to an object. For example, a variable or a pointer to a variable are both lvalues. They are objects where data may be stored. However, expressions such as $a + b$ or $\text{sum}(d,e)$ are not lvalues because they do not refer to a location in memory where data may be stored.
syntax	Rules for writing a computer language that specify how to spell words and punctuate the statements--similar to grammar in the English language.
rvalue	An expression which may appear on the right side of an assignment operator.

Chapter 2

Program Structure

2.1 Functions

C programs are composed of functions. A function is a program module that performs some specific task for the program. Typically, a C program will consist of many small functions, each performing a very specific task.

At a minimum, a C program must contain a function called "main". Program execution begins with the "main" function.

The following example shows the minimum complete C program. It is a program that does nothing.

```
main()
{
}
```

Each function is a stand-alone utility that can be called to perform its specified task. A typical program structure is represented in the following block diagram. The outer block represents a source file that contains C functions.

```

+-----+
! source file
!
!   +-----+
!   ! main()
!   ! {
!   !   /*required once per program*/
!   ! }
!   +-----+
!
!   +-----+
!   ! function_1()
!   ! {
!   !   /*optional*/
!   ! }
!   +-----+
!
!           :
!           :
!           :
!           :
!   +-----+
!   ! function_n()
!   ! {
!   !   /*optional*/
!   ! }
!   +-----+
!
+-----+

```

Although it is possible to put all of the C source code for a program in function "main", splitting the program into smaller, logical units is good programming practice. It makes the program easier to read, understand, and maintain. For large programs, it is also a good practice to split the program into several different source files, each containing one or more functions. Of course, only one of the source files should contain the function called "main". Individual source files may be compiled separately. Therefore, when a function is modified, it is only necessary to compile the source file that contains the modified function.

A function can call another function. A function can even call itself (known as recursion). However, a function cannot define another function within its boundaries. In other words, a function definition cannot be nested. That is why the block diagram above shows non-overlapping function blocks. The following program structure is not allowed.

```

+-----+
! source file                               !
! +-----+                                 !
! ! function 1                             ! !
! ! +-----+                             ! !
! ! ! function 2                           ! ! ! illegal
! ! !                                     ! ! !
! ! +-----+                             ! !
! !                                       ! !
! +-----+                             !
!                                       !
+-----+

```

A function definition is itself composed of several sections: a function header, argument declarations, and a compound statement.

2.1.1 Function Header

The function header is used to name the function, determine the type (int, float, char, etc.) of the value the function returns, and to list the variables (arguments), if any, which are used to store data passed to the function. The form of a function header is as follows.

```
data-type function-name(arg1, arg2, ... argn)
```

Example:

```
char address(name, street, city, state, zip)
```

The above example defines a function named "address" which has 5 arguments and returns a value of type char (character). If the type-specifier (in this case char) is missing, then the default type of the returned value is int (integer).

2.1.2 Argument Declarations

The arguments, enclosed in parentheses, form what is known as the argument list. The argument list is optional. When a function requires no arguments, the argument list is represented as "()". If a function does require arguments, the data type of each argument should be declared. An argument that is not declared is assumed to be of type int. The form of a declaration is as follows:

```
data-type var1, var2, ...varn;
```

The declarations for the arguments of a function immediately follow the function header.

Example:

```
char address(street, city, state, zip)
char    street[], city[], state[];
long    zip;
```

2.1.3 Function Body

The body of the function is a compound statement (statements enclosed by braces, { and }). The body of the function contains the statements that perform the specific task of the function.

Example:

```
char address(street, city, state, zip)
char    street[], city[], state[];
long    zip;
{
    /* statements go here */
}
```

2.2 Local Variables

In addition to the variables (arguments) that are used for passing data into a function, a function may also have variables of its own (local variables). Local variables are private variables that are not accessible from any other function. The declarations for local variables are part of the function body. All local variables must be declared prior to the first executable statement.

Example:

```
int address(street, city, state, zip)
char  street[], city[], state[];
long  zip;          /* argument declarations */
{
    char  c;
    int   i;        /* local variable declarations */
    float f;
    double d;

    /* executable statements */
}
```

2.3 Global Variables

In addition to functions, a source file may contain definitions of global variables. Global variables are variables that are defined outside of any functions in the source file. They may appear anywhere in the source file as long as they are not inside a function. A global variable definition creates a variable that may be used by one or more (perhaps all) of the functions in a program.

Example:

```
char time[8];          /* global variable definitions */
char date[8];

int address(street, city, state, zip)
char  street[], city[], state[];
long  zip;             /* argument declarations */
{
    char  c;
    int   i;           /* local variable declarations */
    float f;
    double d;

    /* statements go here */
}
```

A function may access any global variable that is defined prior to the the definition of the function in the source file. If the global variable definitions appear at the beginning of a source file, then all functions in that source file have access to those global variables. A function may also access a global variable that is defined after the function in the same source file, or defined in a different source file, by declaring the global variable as an external variable. An external declaration does not define a new global variable. It simply allows access to a global variable defined elsewhere. The external declaration may be either inside or outside a function. If the declaration appears outside a function, then all following functions may access the externally declared variable. An external declaration inside a function provides access only for the function containing the declaration.

Example:

```
char time[8];          /* global variable definitions */
char date[8];

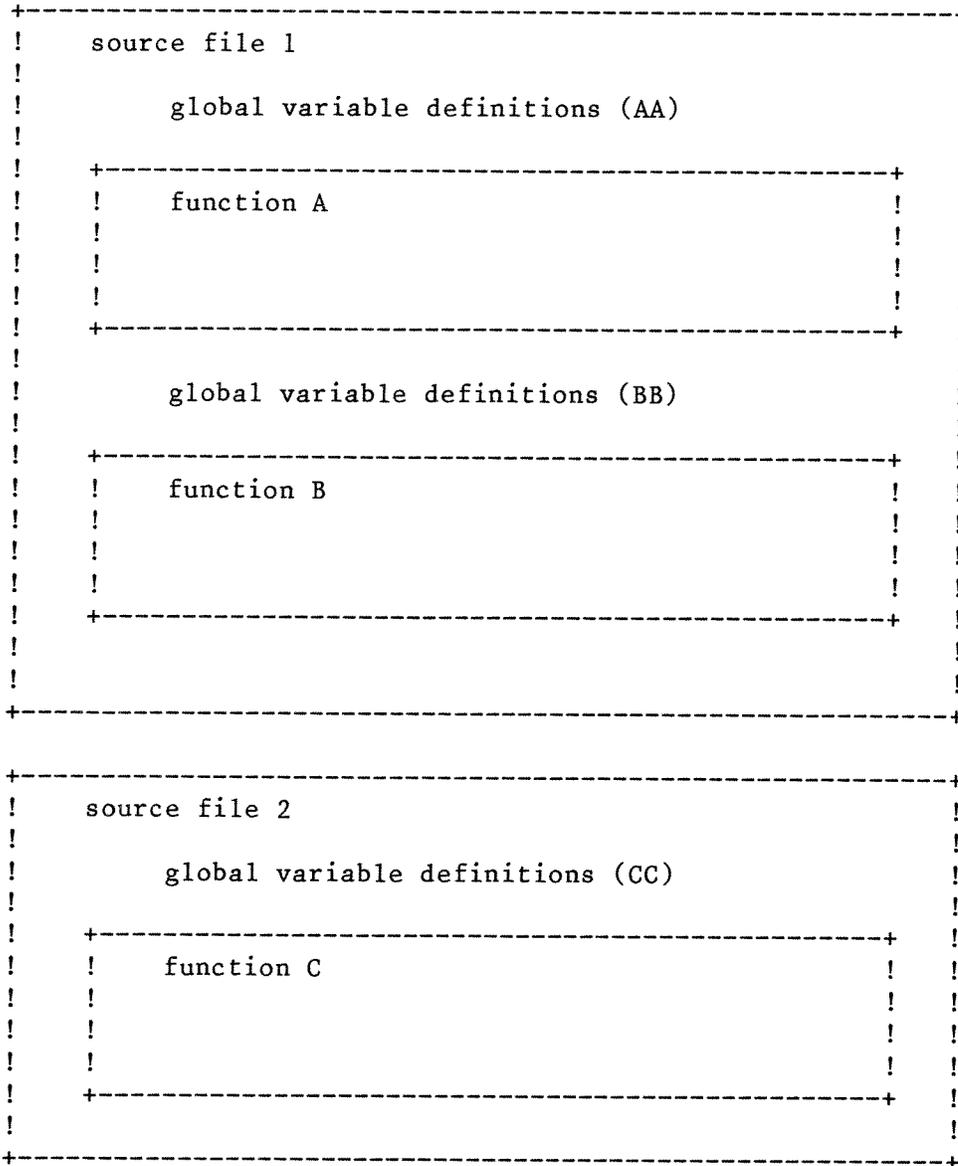
extern float cost;    /* external variable declarations */
extern double total;

int address(street, city, state, zip)
char    street[], city[], state[];
long    zip;          /* argument declarations */
{
    extern int counter; /* external variable declarations */
    extern char *pointer;

    char    c;
    int     i;          /* local variable declarations */
    float   f;
    double  d;

    /* executable statements */
}
```

Consider the following source files which contain functions and global variables.



Functions A and B both have access to the global variable definitions labeled AA. Only function B has access to the global variable definitions labeled BB. If function A needs to have access to the global variable definitions BB, either the BB definitions should be moved in the source file before function A or function A must make external declarations for the global variables BB. If function A needs to access the global variable definitions CC, which are in another source file, then external declarations for the global variables CC must appear prior to function A or inside function A.

Chapter 3

Basic Data Types and Declarations

The basic data types of C are character, integer, and floating point.

3.1 Character Variables

Character variables are used to store single ASCII characters. A character variable is one byte (8 bits) in size, the size necessary to hold one ASCII character. Character variables are declared as:

```
char var1, var2, ... varn;
```

Character variables may be mixed with numeric data types in expressions. The numeric value of a character is between 0 and 255. The ASCII table in the appendix of this manual shows the corresponding numeric value of each character.

Example:

```
main()
{
    char a, b, c;
    a = 'A';          /* 'A' is a character constant */
    b = a + 1;       /* 'B' is equivalent to 'A' + 1 */
    c = 67;          /* 'C' has a decimal value of 67 */
    printf("The decimal value of %c is %d\n",a,a);
    printf("The decimal value of %c is %d\n",b,b);
    printf("The decimal value of %c is %d\n",c,c);
}
```

3.2 Integer Variables

Integer variables are used to store whole numbers (no fractional part). The size of an integer variable is implementation dependent. The System Implementation Manual discusses the size and range of integer variables. Integer variables are declared as:

```
int var1, var2, ... varn;
```

Integer variables may be mixed with other basic data types in expressions.

Example:

```
main() /* print out the range for integer values */
{
    int lower, upper;
    int i, j;
    lower = upper = 0;
    for (i=1; !(upper > 0); i+=2) {
        for (lower = 1, j = 1; j <= i; j++)
            lower = -2 * lower;
        upper = lower - 1;
    }
    printf("integer range: %d to %d",lower,upper);
}
```

3.2.1 Short Integers

There is an adjective (short) that can be used in the declaration of an integer variable that causes the minimum amount of storage to be allocated. Short integers may be used to store positive and negative integer values. The range of values that may be stored in a short integer variable is smaller than that of an integer variable. Short integers are declared as:

```
short int var1, var2, ... varn;
```

The int keyword is optional and may be omitted.

Example:

```
main() /* print out the range for short integer values */
{
    short lower, upper;
    int i, j;
    lower = upper = 0;
    for (i=1; !(upper > 0); i+=2) {
        for (lower = 1, j = 1; j <= i; j++)
            lower = -2 * lower;
        upper = lower - 1;
    }
    printf("short integer range: %d to %d",lower,upper);
}
```

3.2.2 Long Integers

There is an adjective (long) that can be used in the declaration of an integer variable that causes the maximum amount of storage to be allocated. Long integers may be used to store positive and negative integer values. The range of values that may be stored in a long integer variable is larger than that of an integer variable. Long integers are declared as:

```
long int var1, var2, ... varn;
```

The int keyword is optional and may be omitted.

Example:

```
main() /* print out the range for long integer values */
{
    long lower, upper;
    int i, j;
    lower = upper = 0;
    for (i=1; !(upper > 0); i+=2) {
        for (lower = 1, j = 1; j <= i; j++)
            lower = -2 * lower;
        upper = lower - 1;
    }
    lower++; /* largest negative long prints as -0 */
    printf("long integer range: %ld to %ld",lower,upper);
}
```

3.2.3 Unsigned Integers

There is an adjective (unsigned) that can be used in the declaration of an integer variable that allocates the same amount of storage as a normal integer. However, unsigned integers may be used to store only positive integer values. The maximum positive value of an unsigned integer is approximately twice that of a normal integer. Unsigned integers are declared as:

```
unsigned int var1, var2, ... varn;
```

The int keyword is optional and may be omitted.

Example:

```
main() /* print out the range for unsigned integer values */
{
    unsigned lower, upper, save;
    int      i, j;
    lower = 0;
    upper = 1;
    for (i=1; upper != 0; i++)
        for (upper = 1, j = 1; j <= i; j++) {
            save = upper;
            upper = 2 * upper;
        }
    upper = 2L * save - 1; /* L forces long calculation */
    printf("unsigned integer range: %u to %u",lower,upper);
}
```

3.3 Floating Point Variables

Floating point variables are used for numbers that may have a fractional part. A floating point number consists of two parts: the mantissa and the exponent. The size of the mantissa determines the number of digits of accuracy while the exponent size determines the range of floating point values. The accuracy and range of floating point variables is discussed in the System Implementation Manual. Single precision floating point variables are declared as:

```
float var1, var2, ... varn;
```

Example:

```
main()
{
    float    cost_per_item, total_cost;
    int      number_of_items;
    cost_per_item = 1.23;
    number_of_items = 3;
    total_cost = number_of_items * cost_per_item;
    printf("Total cost = %.2f", total_cost);
}
```

3.3.1 Double Precision

Double precision variables allow more digits to be stored in the mantissa of a floating point number. When maximum accuracy is needed, then double precision floating point variables should be used. Double precision variables are declared as:

```
double float var1, var2, ... varn;
```

The float keyword is optional and may be omitted.

Example:

```
main()
{
    double radius, pi, area, circumference;
    pi = 3.141592654;
    printf("Enter the radius of a circle:");
    scanf("%lf", &radius);
    area = pi * radius * radius;
    circumference = 2 * radius * pi;
    printf("    The radius is %.10f\n\
    The area is %.10f\n\
    The circumference is %.10f",
    radius, area, circumference);
}
```

3.4 Storage Classes

In addition to having a defined type, all variables have a storage class. A storage class defines how a variable is created and accessed. There are four storage classes, auto, extern, static, and register. The format for declaring a variable of a specific storage class follows:

```
storage_class data_type var1, var2, ... varn;
```

When a storage class is specified, the data_type may be omitted. The compiler will assume the type int if the data_type is omitted.

3.4.1 Auto Variables

The most frequently used storage class is auto. Auto variables are local variables. They are declared inside a function and are accessible only to that function. The use of auto variables minimizes the amount of memory required by a program. The reason is that auto variables are not allocated storage until the function is called. When the function is exited, the storage allocated to the auto variables is released. This dynamic allocation of memory means that auto variables do not retain their values between function calls. They must be initialized inside the function before they are used. Auto variables are declared using the following format:

```
auto data_type var1, var2, ...varn;
```

By default, all variables declared inside a function are auto variables. Therefore, the auto keyword is optional and may be omitted.

Example: /

```
main() /* example of auto (local) variables */
{
    auto char a, b; /* keyword auto is optional */
    auto int i, j; /* either auto or int may be omitted */
}
```

3.4.2 Extern Variables

The storage class `extern` (external) is used to allow access to a variable that has been defined outside a function. A variable defined outside a function is a global variable. Unlike `auto` (local) variables, global variables retain their values at all times. One or more functions may have access to a global variable. A function is allowed access by including an `extern` declaration for the global variable. An `extern` declaration does not allocate any new storage space, it simply references the space allocated to the global variable which has presumably been defined elsewhere. `Extern` variables are declared using the following format:

```
extern data_type var1, var2, ...varn;
```

A function may reference a global variable that has been defined in the same source file or in a different source file. In either case, simply make an `extern` declaration (inside the function) for the global variable. In the case that the global variable is defined in the same source file and prior to the function, the `extern` declaration may be omitted. In such a case, the `extern` declaration is implicitly made by the compiler.

If all the functions in one source file need to access a global variable defined in a different source file, a single global `extern` declaration will provide all subsequent functions access to the global variable.

Example:

```
Source File A
-----

char      GlA;      /* define global variable */
int       G2A;      /* define global variable */
extern long GlB;    /* external declaration for
                    global variable defined
                    in source file B      */

main()
{
    float L1A;
    double L2A;

    /* main can access global variables GlA, G2A,
       and GlB as well as local variables L1A and
       L2A                                     */
}

Afunction()
{
    extern char GlA;
    extern float G2B;

    /* Afunction can access global variables GlA,
       G2A, GlB, and G2B                       */
}

Source File B
-----

long      GlB;      /* define global variable */
float     G2B;      /* define global variable */

Bfunction()
{
    extern char GlA;

    /* Bfunction can access global variables GlA,
       GlB, and G2B                             */
}
```

3.4.3 Static Variables

The storage class `static` may be used with either global or local variables.

Static variables that are declared inside a function, like auto variables, are accessible only to that function. The difference between local static and auto variables is in the way storage is allocated. A static variable is allocated permanent storage, like global variables. Therefore, a local static variable retains its value between function calls. The static variable may be defined during the first call to the function, and subsequent calls may use this defined value.

Example:

```
setxy(flag)
int flag;
{
    static float x, y;
    if (flag) {
        x = 10;
        y = 10;
    }
    else {
        x = x + .5;
        y = y + .5;
    }
}
```

A global variable that is declared to be static is similar to a normal global variable. The difference is that a static global variable cannot be accessed by a function in another source file. A global static variable is invisible to all functions except those following it in the same source file. A global static is normally used when a group of functions must share a common variable and you want to make the variable invisible to functions in other source files.

Source File A

```
main()
{
    extern double seed; /* this will not allow access to the
                        global static variable, seed, in
                        source file B */
    double random();
    randomize(65.2);
    seed = checkseed();
}
```

Source File B

```
static double seed; /* invisible to all
                    other source files */

randomize(value)
double value;
{
    seed = value;
}

double checkseed()
{
    return seed;
}
```

3.4.4 Register Variables

The register storage class may be used only for local variables (declared inside a function). The register storage class tells the compiler to make these variables as efficient to use as possible. Normally, this involves allocating a specific hardware register for storing the variable. The actual effect of the register storage class is system dependent. See the Systems Implementation Manual.

Example:

```
main()
{
    register int i;
    int a[1000];
    for (i=0; i<1000; i++) a[i] = i;
}
```

3.5 Initialization of Basic Data Types

A declaration may specify an initial value for a variable. A basic data type may be initialized by following the variable name with an equal sign (=), followed by an expression. The expression may contain constants and/or previously declared variables.

Both local and global variables may be initialized. If a global variable is not explicitly initialized, it will have a value of 0. If a local variable is not initialized, it will normally have an undefined value. However, a compiler option provides the ability to automatically initialize local variables to 0.

Example:

```
#define FIRST 10
#define LAST 100

char terminator = 'z';
int size       = LAST - FIRST + 1;
int next       = FIRST + size;

main()
{
    float mph = 1.7e2;
    double degrees = 36.0;
    printf("%c %d %d %f %f", terminator, size,
           next, mph, degrees);
}
```

3.6 Type Definitions

Type definitions allow names to be associated with data types. The reserved word, typedef, is used to define a name for a data type. A typedef associates an identifier with one of the basic data types or with a user defined (composite) data type. The identifier may then be used as a data type in subsequent variable declarations. The format of a typedef is:

```
typedef data_type identifier
```

Typedef provides a convenient way of declaring a type for certain variables of a program that may need to be changed to another type at some later time.

If an alias identifier is used as a type, rather than a predefined type, then changing the type at a later time simply involves changing the typedef.

Example:

```
typedef int  COUNTERS;
typedef float SUMS;

main()
{
    COUNTERS i, j;
    SUMS     total;
    int      number;
    float    length;
    /* do something here */
}
```

In the previous example, if you decided that all variables declared as type COUNTERS needed to be long integers and all variables declared as SUMS needed to be double precision, you would simply change the two typedefs (change int to long and float to double) and recompile. A good practice is to uppercase the typedef identifiers so that they are easily distinguished from other declarations.

You can also use typedef to define names for user defined types. For example, it is often convenient to define a name for a structure definition. The name can then be used to declare variables rather than using the structure definition itself.

Example:

```
typedef struct {
    long    part_number;
    unsigned quantity;
    float   cost;
} INVENTORY;

main()
{
    INVENTORY item[100];
    .
    .
    .
}
```

Chapter 4

Basic Operators and Expressions

4.1 Operator Precedence and Grouping

An expression consists of an identifier or a sequence of identifiers and operators. An identifier used with an operator is called an operand. Two things effect the order in which operations are performed in an expression.

- (1) precedence of operators
- (2) grouping (associativity) of operators.

When several operators are in one expression, some of the operators are acted on before others. The operators acted on first are said to have a higher precedence than those that are acted on later. The higher the precedence, the sooner the operator will be acted on during evaluation of the expression.

There are 15 levels of operator precedence in C. The table at the end of this introduction shows the operators and the precedence level of each operator. The highest precedence level is labeled 1 and the lowest level is labeled 15. Each of these operators is covered in Chapters 4 and 5 and their precedence level is designated by the number shown in the table. What the table shows is that an addition operator at level 4 will be acted on before the relational operator \leq at level 6 or the assignment operator $=$ at level 14. In other words, operators at level 1 will be operated on first, operators at level 2 will be operated on second, etc. through level 15.

Parentheses may be used in expressions to control the order of operations. Parentheses force the enclosed operations to be performed before the operator precedence and grouping rules are applied. If parentheses are nested (parentheses enclosed by other parentheses), then the innermost parenthesized operations are performed first. Since there are so many operators at different precedence levels in C, you may want to make liberal use of parentheses to insure that an expression is evaluated as intended.

Example: $a = b + 2 <= c$ evaluates as $a = ((b + 2) <= c)$

Notice that several of the various precedence levels have more than one operator shown for the level. Operators shown on the same level have equal precedence. For example, the operators $<$, $<=$, $>=$, and $>$ all have equal precedence at level 6. Since all have equal precedence, the grouping (associativity) rule for evaluation of expressions comes into play. Grouping occurs from left to right or from right to left, depending on the operators. Most operators group (associate) left to right, but some such as the unary operators group right to left. Take the following expression to demonstrate grouping

$$a <= b > c < d >= e$$

The relational operators all have equal precedence, so the grouping rule takes effect. The relational operators group from left to right. This means that the leftmost operator is acted on first. The rightmost operator is acted on last.

$$(((a <= b) > c) < d) >= e)$$

For grouping right to left, try

$$a = b += c *= d \text{ which groups as } a = (b += (c *= d))$$

When expressions are evaluated, the precedence rule is applied first, followed by the grouping rule. As an example, consider the following expression that contains operators at different precedence levels.

$$a = b + c / d * 2 - e$$

The operators, $/$ and $*$, have the highest precedence. These operations will therefore be performed first. But since they have equal precedence, which will be performed first? The grouping rule must decide. Since the $/$ and $*$ operators group left to right, the division is performed first, followed by the multiplication. Using parentheses to illustrate, the following expression is equivalent.

$$a = b + ((c / d) * 2) - e$$

The operators, $+$ and $-$, have the next highest precedence. Again, these operators have equal precedence so the grouping rule must be applied. Since the $+$ and $-$ operators group left to right, the addition is performed first. The following expression is equivalent.

$$a = ((b + ((c / d) * 2)) - e)$$

The only operator left is the assignment operator (=). This is the last operation performed. The following parenthesized expression is equivalent to the original expression.

$$(a = ((b + ((c/d) * 2)) - e))$$

The following table shows the precedence levels of the operators and the order of evaluation (grouping) of each. Operators on the same line have the same precedence level, so <, >, <=, and >=, for example, all have the same precedence. Each line is in order of precedence. Primary expression operators have the highest precedence and the comma operator has the lowest precedence. Operators at precedence level 1 are operated on first, then operators at precedence level 2, and so on through precedence level 15. The numbers to the left of the operators are used throughout Chapters 4 and 5 to show the precedence level during discussions on the individual operators.

Name	Lvl	Operators	Grouping
primary	1	() . [] ->	L to R
unary	2	! ~ - (typename) * & ++ -- sizeof	R to L
binary	3	* / %	Left to Right
arithmetic	4	+ -	Right to Left
shift	5	<< >>	L to R
relational	6	< <= >= >	L to R
equality	7	== !=	L to R
bitwise	8	&	Left to Right
logical	9	^	Right to Left
logical	10		Left to Right
connective	11	&&	Left to Right
connective	12		Right to Left
conditional	13	?:	R to L
assignment	14	= += -= /= *= %= <<= >>= &= ^= =	R to L
comma	15	,	L to R

4.2 Assignment Operator

The = operator assigns (stores) a value to a memory location. The assignment operator requires two operands. The left operand is an lvalue, the location in memory where a value may be stored. The lvalue most often used is a simple variable name. The right operand is an expression.

Examples:

```
c = 'a';      /* assigns the character a to variable c */
i = 23;       /* assigns the integer 23 to variable i */
f = 32.8;     /* assigns the real 32.8 to variable f */
```

4.3 Arithmetic Operators

The arithmetic operators require two operands.

Example:

operand	operator	operand
length	*	width
x	/	y
n	%	8
a[i]	+	a[i + 2]
big	-	little

The following table lists all the arithmetic operators, the precedence level of each, the operations they perform, the type of operands which may be used, and the type of the result each operator yields. Other properties of each operator are given in the following paragraphs.

Op	Lvl	Operation	Type of Operand	Type of Result
*	3	multiplication	basic data type	basic data type
/	3	division	basic data type	basic data type
%	3	modulo	integer	integer
+	4	addition	basic data type or pointer	basic data type or pointer
-	4	subtraction	basic data type or pointer	basic data type or pointer

4.3.1 Properties of + Operator

The + operator performs addition between two operands of any of the basic data types, or between a pointer and an integer. The + operator groups left to right.

Example:

$x = y + z + \text{incr}$ groups as $x = ((y + z) + \text{incr})$

4.3.2 Properties of - Operator

The - operator performs subtraction between two operands of any basic data type, or between a pointer and an integer. The - operator may also be used to subtract 2 pointers if the pointers both point to the same array. The result of the subtraction yields the number of array elements between the two pointers. The - operator groups left to right.

Example:

`x = y - z - incr` groups as `x = ((y - z) - incr)`

The `-` operator may also be used as a unary operator, requiring only one operand. In this case, the operand must be a basic data type. When used as a unary operator, the result is the negative of the value of the operand. If the operand is positive, the result is negative. If the operand is negative, the result is positive.

Examples:

`-10` `-32.3` `-a` `-number`

4.3.3 Properties of `*` Operator

The `*` operator performs multiplication between two operands of any of the basic data types. Multiplication cannot be performed using pointers. The `*` operator groups left to right.

Example:

`x = y * z * incr` groups as `x = ((y * z) * incr)`

4.3.4 Properties of `/` Operator

The `/` operator performs division with operands of any of the basic data types. Division may not be performed on pointers. Division with integer operands yields an integer type result which is truncated. The `/` operator groups left to right.

Example:

`x = y / z / incr` groups as `x = ((y / z) / incr)`

If `a` and `b` are of type `INT`, then

`(a / b) * b`

does not always equal "`a`" since integer division truncates the fractional part. However,

`((a / b) * b) + (a % b) == a`

is always true. The `%` operator adds the remainder of the integer division, the part that was truncated.

4.3.5 Properties of `%` Operator

The `%` operator performs modulo arithmetic on integer operands. The `%` operator yields the remainder of the division of the first operand by the second operand. The `%` operator groups left to right.

Example:

`x = y % z % incr` groups as `x = ((y % z) % incr);`

<code>32 % 11 = 10</code>	<code>27 % 7 = 6</code>
<code>15 % 3 = 0</code>	<code>3 % 5 = 3</code>

4.4 Relational and Equality Operators

The relational and equality operators are used to test or compare the condition between two operands. Both relational and equality operators require two operands. The operands may be basic data types or pointers. The relational operators are `<`, `>`, `<=`, and `>=`. The equality operators are `==` and `!=`. Relational and equality operators produce an integer result that is non-zero if the condition is true, or 0 if the condition is false. For example, the expression `5 > 7` evaluates to 0 since the condition is false, 5 is not greater than 7.

The following table lists the relational and equality operators, the precedence level of each operator, the conditions they test, the type of operands which may be used, and the type of the result. Any other properties for each operator are given in the paragraphs following the table.

Op	Lvl	Operation	Type of Operand	Type of Result
<	6	less than	basic data type or pointer	integer
<=	6	less than or equal to	basic data type or pointer	integer
>	6	greater than	basic data type or pointer	integer
>=	6	greater than or equal to	basic data type or pointer	integer
==	6	equal to	basic data type or pointer	integer
!=	6	not equal to	basic data type or pointer	integer

Examples of Relational and Equality Operators:

expression	result	order of evaluation
2 < 3	1 (true)	(2 < 3)
3 <= 2	0 (false)	(3 <= 2)
2 * 3 >= 6	1 (true)	(2 * 3) >= 6
3 != 2	1 (true)	(3 != 2)
3 != 6 / 2	0 (false)	3 != (6 / 2)
1 == 6 % 5	1 (true)	1 == (6 % 5)
4 > 8 * 23 % 10	0 (false)	4 > ((8 * 23) % 10)
16 > 8 == 1	1 (true)	(16 > 8) == 1
1 != (7 == 0)	1 (true)	1 != (7 == 0)
1 != 7 == 0	0 (false)	(1 != 7) == 0

The equality operators, == and !=, may be used to compare any two pointer variables. However, the relational operators, <, >, <=, and >= may be used to compare only pointers that point to elements of same array. When comparing pointers with the relational operators, the result is true if the left operand points to a lower numbered element than the right operand. Otherwise the result is false.

4.4.1 Properties of < Operator

The < operator yields a non-zero value (true) if the left operand is less than the right operand. Otherwise, it yields a 0 (false). The < operator may be used to compare operands that are basic data types, or pointers to elements of an array. The < operator groups left to right.

4.4.2 Properties of <= Operator

The <= operator yields a non-zero value (true) if the left operand is less than or equal to the right operand. Otherwise, it yields a 0 (false). The <= operator may be used to compare operands that are basic data types, or pointers to elements of an array. The <= operator groups left to right.

4.4.3 Properties of > Operator

The > operator yields a non-zero value (true) if the left operand is greater than the right operand. Otherwise, it yields a 0 (false). The > operator may be used to compare operands that are basic data types, or pointers to elements of an array. The > operator groups left to right.

4.4.4 Properties of >= Operator

The >= operator yields a non-zero value (true) if the left operand is greater than or equal to the right operand. Otherwise, it yields a 0 (false). The >= operator may be used to compare operands that are basic data types, or pointers to elements of an array. The >= operator groups left to right.

4.4.5 Properties of == Operator

The == operator yields a non-zero value (true) if the left operand is equal to the right operand. Otherwise, it yields a 0 (false). The == operator may be used to compare operands that are basic data types, or pointers. The == operator groups left to right.

4.4.6 Properties of != Operator

The != operator yields a non-zero value (true) if the left operand is not equal to the right operand. Otherwise, it yields a 0 (false). The != operator may be used to compare operands that are basic data types, or pointers. The != operator groups left to right.

4.5 Logical Operators

The relational and equality operators produce a value that is either non-zero or 0. These are often referred to as logical values, a non-zero value representing true, and a 0 value representing false. The logical operators also produce logical values. There are three logical operators, the && (and) operator, the || (or) operator, and the ! (not) operator.

The logical operators, the precedence level of each operator, the operations they perform, the type of operands and the type of the result are shown in the following table. The properties of each operator are given in the paragraphs after the table.

Op	Lvl	Operation	Type of Operand	Type of Result
		logical		
&&	11	connective	basic data type	integer
		AND		
		logical		
	12	connective	basic data type	integer
		OR		
		logical		
!	2	negate	basic data type	integer
		NOT		

4.5.1 Properties of && Operator

The && (logical and) operator requires two operands which represent logical values, 0 (false) or non-zero (true). The result of the operation is non-zero (true), if and only if both operands are non-zero (true). Otherwise the result is 0 (false).

Truth Table for &&

condition	result
false && false	false
false && true	false
true && false	false
true && true	true

4.5.2 Properties of || Operator

The || (logical or) operator requires two operands which represent logical values, 0 (false) or non-zero (true). The result of the operation is 0 (false), if and only if both operands are 0 (false). Otherwise the result is non-zero (true).

Truth Table for ||

condition	result
false false	false
false true	true
true false	true
true true	true

4.5.3 Properties of ! Operator

The ! (logical negate) operator requires one operand which represents a logical value, 0 (false) or non-zero (true). The result of the operation is non-zero (true), if and only if the operand is 0 (false). Otherwise the result is 0 (false).

Truth Table for !

condition	result
!false	true
!true	false

4.6 Type Conversions

C supports the use of mixed mode arithmetic. Mixed mode arithmetic is the use of more than one data type in an expression. For example, multiplying a float by an int. C allows this by performing implicit type conversions before evaluating the expression. The following rules apply to implicit data type conversion in expressions. The rules are applied to all expressions in the order listed.

- (1) All char and short operands are converted to int and all float operands are converted to double. Proceed to step 2.
- (2) If there is a double operand, then all other operands are converted to double and the result is double. Otherwise proceed to step 3.
- (3) If there is a long operand, then all other operands are converted to long and the result is long. Otherwise proceed to step 4.
- (4) If there is an unsigned operand, then all other operands are converted to unsigned and the result is unsigned. Otherwise proceed to step 5.
- (5) All operands are int and the result is int.

The above rules always cause an upward type conversion. That is, all operands are converted to a type that is more precise. Therefore, the application of these rules increase the accuracy of the result.

There are also implicit downward type conversions in C. These type conversions take place when the result of an expression is assigned to a variable that is less precise than the result. Another time that this can occur is when the type of a function is less precise than the return value. Downward type conversions can result in loss of accuracy because the value is truncated to fit the type of its destination.

result type	destination type	conversion rule
double	float	round and truncate excess bits
float	long	truncate fractional part
long	int	truncate high order bits
int	short	truncate high order bits
short	char	truncate high order bits

Chapter 5

More Operators and Expressions

5.1 Increment and Decrement Operators

Increment and decrement operators require a single operand. The operand must be an lvalue. That is, it must refer to a memory location where a value may be stored. For example, a variable name is an lvalue. The operand may be a basic data type or a pointer. Incrementing a basic data type is equivalent to adding 1 to it. Decrementing a basic data type is equivalent to subtracting 1 from it. When a pointer type is incremented or decremented, the value that it is added or subtracted is not necessarily 1. It is equal to the size (in bytes) of the pointed to data type. Incrementing a variable that points to a character (char) will add 1 to the pointer since the size of a character is 1 byte. Decrementing a pointer to a character will subtract 1 from the pointer.

Besides the assignment operator, the increment and decrement operators are the only operators that alter the value of an operand. The following table lists the increment and decrement operators, the precedence level of each operator, the operations they perform, the type of operands which may be used, and the type of the result. Any other properties of each of the operators is given in the paragraphs following the table.

Op	Lvl	Operation	Type of Operand	Type of Result
			basic data type or pointer	basic data type or pointer
++	2	increment		
			basic data type or pointer	basic data type or pointer
--	2	decrement		

Examples of Increment and Decrement Operators:

expressions	order of evaluation
a++	(a++)
++a	(++a)
p++ + ++b	((p++)+(++b))
z / --c	z / (--c)
c-- % 3	((c--) % 3)
z / c--	(z / (c--))

5.1.1 Properties of ++ Operator

The increment operator may be used to increment a basic data type or a pointer. The operand must be an lvalue. The increment operator may precede the operand (pre-increment) or follow the operand (post-increment).

pre-increment: ++a post-increment: a++

When pre-increment is used, the operands value is incremented and stored as its new value. The new value is the result of the operation. When post-increment is used, the operands value is incremented and stored as the new value. However, the result of the operation is the old value, the value of the operand before being incremented. The ++ operator groups right to left.

Example: assume s and t are integers, t = 5

	result
s = ++t;	t = 6, s = 6
s = t++;	t = 6, s = 5

5.1.2 Properties of -- Operator

The decrement operator may be used to decrement a basic data type or a pointer. The operand must be an lvalue. The decrement operator may precede the operand (pre-decrement) or follow the operand (post-decrement).

pre-decrement: --a post-decrement: a--

When pre-decrement is used, the operands value is decremented and stored as its new value. The new value is the result of the operation. When post-decrement is used, the operands value is decremented and stored as the new value. However, the result of the operation is the old value, the value

of the operand before being decremented. The `--` operator groups right to left.

Example: assume s and t are integers, t = 5

	result

s = --t;	t = 4, s = 4
s = t--;	t = 4, s = 5

5.2 Bitwise Operators

The bitwise operators produce a result that is determined by examining each bit of the operand(s). The `~` (bitwise negate) operator requires only one operand. All the rest require two operands. The following table lists all of the bitwise operators, the precedence level of each operator, the operations they perform, the type of operands which may be used, and the results of the operators.

Op	Lvl	Operation	Type of Operand	Type of Result
		bitwise	basic data type	same as operand
<code>~</code>	2	negate	(excluding float & double)	(except char --> int)
		bitwise	basic data type	same as operand
<code>>></code>	5	right shift	(excluding float & double)	(except char --> int)
		bitwise	basic data type	same as operand
<code><<</code>	5	left shift	(excluding float & double)	(except char --> int)
		bitwise AND	basic data type	same as operand
<code>&</code>	8	bitwise AND	(excluding float & double)	(except char --> int)
		bitwise exclusive OR	basic data type	same as operand
<code>^</code>	9	exclusive OR	(excluding float & double)	(except char --> int)
		bitwise inclusive OR	basic data type	same as operand
<code> </code>	10	inclusive OR	(excluding float & double)	(except char --> int)

5.2.1 Properties of `~` Operator

The `~` (bitwise negate) operator requires one operand. The result of the operation is the one's complement of the operand. The `~` operator changes all 1 bits to 0 and all 0 bits to 1. The `~` operator groups right to left. In the following examples, S stands for int (which is signed) and U stands for unsigned.

Examples:

	decimal	binary representation	binary	result decimal
S	~3	~00000011	11111100	- 4
U	~3	~00000011	11111100	252
S	~9	~00001001	11110110	- 10
U	~9	~00001001	11110110	246
S	~110	~01101110	10010001	-111
U	~110	~01101110	10010001	145

Notice that $\sim(\sim 3) = 3$ and $\sim(\sim 110) = 110$.

5.2.2 Properties of >> Operator

The >> (shift right) operator requires two operands. The left operand is shifted to the right by the number of bits specified by the right operand. If the left operand is an unsigned integer, then the vacated bits in the left operand are 0 filled (also known as logical fill). If the left operand is a signed integer, then the vacated bits in the left operand are filled with an extension of the sign bit (also known as arithmetic fill). The >> operator groups right to left. In the following examples, S stands for int (which is signed) and U stands for unsigned.

	decimal	binary representation	binary	result decimal
S	110 >> 2	01101110 >> 2	00011011	27
U	110 >> 2	01101110 >> 2	00011011	27
S	-42 >> 3	11010110 >> 3	11111010	-5
U	214 >> 3	11010110 >> 3	00011010	26
S	-72 >> 1	10111000 >> 1	11011100	-36
U	184 >> 1	10111000 >> 1	01011100	92
S	-42 >> 8	11010110 >> 8	11111111	-1
U	214 >> 8	11010110 >> 8	00000000	0

5.2.3 Properties of << Operator

The << (shift left) operator requires two operands. The << operator shifts the left operand to the left by the number of bits specified by the right operand. The vacated bits in the left operand are 0 filled (logical fill). The << operand groups right to left. In the following examples, S stands for int (which is signed) and U stands for unsigned.

	decimal	binary representation	result binary	decimal
S	110 << 2	01101110 << 2	10111000	-72
U	110 << 2	01101110 << 2	10111000	184
S	-42 << 3	11010110 << 3	10110000	-80
U	214 << 3	11010110 << 3	10110000	176
S	-72 << 1	10111000 << 1	01110000	112
U	184 << 1	10111000 << 1	01110000	112
S	-42 << 8	11010110 << 8	00000000	0
U	214 << 8	11010110 << 8	00000000	0

5.2.4 Properties of & Operator

The & operator performs a bitwise AND between two operands. This operator compares each bit of the left operand with the corresponding bit of the right operand. For each bit, the result is 1 if the two compared bits are both 1, otherwise the result is 0.

Table for bitwise &

```

0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

```

The & operator groups left to right. In the following examples, S stands for int (which is signed), U for unsigned, and B for the binary representation of the decimal value.

S 2 & 1 --- 0	U 2 & 1 --- 0	B 00000010 & 00000001 ----- 00000000
S 2 & 3 --- 2	U 2 & 3 --- 2	B 00000010 & 00000011 ----- 00000010
S 3 & 5 --- 1	U 3 & 5 --- 1	B 00000011 & 00000101 ----- 00000001
S -72 & 110 ----- 40	U 184 & 110 ----- 40	B 10111000 & 01101110 ----- 00101000

5.2.5 Properties of ^ Operator

The ^ operator performs a bitwise XOR (exclusive OR) between two operands. This operator compares each bit of the left operand with the corresponding bit of the right operand. For each bit, the result is 0 if the two compared bits have the same value, otherwise the result is 1.

Table for bitwise ^

0 ^ 0 = 0
 0 ^ 1 = 1
 1 ^ 0 = 1
 1 ^ 1 = 0

The ^ operator groups left to right. In the following examples, S stands for int (which is signed), U for unsigned, and B for the binary representation of the decimal value.

S	2	U	2	B	00000010
	^ 1		^ 1		^ 00000001
	---		---		-----
	3		3		00000011
S	2	U	2	B	00000010
	^ 3		^ 3		^ 00000011
	---		---		-----
	1		1		00000001
S	3	U	3	B	00000011
	^ 5		^ 5		^ 00000101
	---		---		-----
	6		6		00000110
S	-72	U	184	B	10111000
	^ 110		^ 110		^ 01101110
	-----		-----		-----
	-42		214		11010110

5.2.6 Properties of | Operator

The | operator performs a bitwise OR (inclusive OR) between the two integer operands. This operator compares each bit of the left operand with the corresponding bit of the right operand. For each bit, the result is 0 if the two compared bits are both 0, otherwise the result is 1.

Table for bitwise |

0		0	=	0
0		1	=	1
1		0	=	1
1		1	=	1

The | operator groups left to right. In the following examples, S stands for int (which is signed), U for unsigned, and B for the binary representation of the decimal value.

S	2	U	2	B	00000010
	1		1		00000001
	---		---		-----
	3		3		00000011
S	2	U	2	B	00000010
	3		3		00000011
	---		---		-----
	3		3		00000011
S	3	U	3	B	00000011
	5		5		00000101
	---		---		-----
	7		7		00000111
S	-72	U	184	B	10111000
	110		110		01101110
	-----		-----		-----
	-2		254		11111110

5.3 Assignment Operators

In addition to the simple assignment operator =, there are operators that are combinations of the assignment operator with either an arithmetic or bitwise operator. These combination operators provide a short hand notation for assignment.

lvalue op= expr short hand for: lvalue = lvalue op expr

where lvalue is normally a variable name
 op is one of the arithmetic or bitwise operators
 expr is any valid C expression

The assignment operators, the precedence level of each operator, the operation they perform, the type of operand they require, and the type of the result are shown in the following table.

Op	Lvl	Operation	Type of Operand	Type of Result
+=	14	addition assignment	basic data type or pointer	type of left operand
-=	14	subtraction assignment	basic data type or pointer	type of left operand
*=	14	multiplication assignment	basic data type	type of left operand
/=	14	division assignment	basic data type	type of left operand
%=	14	modulo assignment	basic data type (excluding float & double)	type of left operand
>>=	14	right shift assignment	basic data type (excluding float & double)	type of left operand
<<=	14	left shift assignment	basic data type (excluding float & double)	type of left operand
&=	14	bitwise AND assignment	basic data type (excluding float & double)	type of left operand
=	14	bitwise OR assignment	basic data type (excluding float & double)	type of left operand
^=	14	bitwise XOR assignment	basic data type (excluding float & double)	type of left operand

Examples:

short hand		equivalent
x += 1	====>	x = x + 1
y *= z - delta	====>	y = y * (z - delta)
t %= r	====>	t = t % r
a >>= b	====>	a = a >> b

5.4 Address Of and Contents Of Operators

There are two operators that are related to pointers. Both require one operand. The & (address of) operator returns the address of its operand. The * (contents of) operator returns the contents of the object pointed to by its operand. The & operator groups right to left.

Op	Lvl	Operation	Type of Operand	Type of Result
&	2	address of	variable	pointer to variable
*	2	contents of	pointer to variable	value of variable

5.4.1 Properties of & Operator

The & (address of) operator requires one operand that must be a variable name. The name may refer to a simple variable (basic data type), an array element, a structure, or a member of a structure. The result of the operation is the address of the operand.

Examples:

<u>operation</u>	<u>result</u>
<code>&i</code>	address of integer variable <code>i</code>
<code>&t[2]</code>	address of element 3 of array <code>t</code>
<code>&list.age</code>	address of member <code>age</code> of structure <code>list</code>

5.4.2 Properties of * Operator

The * (contents of) operator requires one operand that must be the address of a variable. The operand may be a single pointer variable or any legal pointer expression. The result of the operation is the value that is stored at that address. The expression, `*(&i)`, reads "contents of the address of `i`". The `&i` is a pointer expression that returns the address of variable `i`. The * operator then returns the value stored at that address. The simple expression, `i`, is therefore equivalent to `*(&i)`.

Pointer variables are declared by preceding the variable name with the * operator. The declaration, `int *ptr`, declares the variable `ptr` as a pointer to a value of type `int`. Assuming `i` is a variable of type `int`, `ptr = &i`; assigns the address of `i` to the variable `ptr`. Then the expression, `*ptr`, returns the value of `i`. The * operator groups right to left.

Example:

```
main()
{
    int i;
    char *ptr;      /* pointer to value of type char */
    char alphabet[27];
    ptr = alphabet; /* equivalent to ptr = &alphabet[0] */
    for (i=0; i<26; i++) *(ptr+i) = i + 'a';
    alphabet[26] = '\0';
    printf("%s",alphabet)
}
```

5.5 Sizeof Operator

The `sizeof` operator requires a single operand that may be either an expression or a data type name. The result of the operation is the size, in bytes (a byte is the amount of memory required to store a single character), of the operand. If the operand is a data type name (eg. `char`), it must be enclosed by parentheses. The two forms for using the `sizeof` operator are as follows.

```
sizeof (type-name)
sizeof expression
```

The "sizeof(type-name)" returns the size of an object of the specified type. For example, sizeof(char) returns the value 1 since a character requires 1 byte of storage.

The "sizeof expression" returns the size of the result of the expression. Any valid expression may be used. For example, sizeof a + 1.0 will return the size of a double floating point value.

Example:

```
main()
{
    char a[10];
    int i;
    struct parts {
        char part1;
        float part2;
    } key;
    printf("size of char   = %d\n", sizeof(char));
    printf("size of int    = %d\n", sizeof i);
    printf("size of long   = %d\n", sizeof(long));
    printf("size of float  = %d\n", sizeof(float));
    printf("size of double = %d\n", sizeof(double));
    if (sizeof(struct parts) != sizeof key)
        printf("this will not be printed");
}
```

5.6 Cast Operator

The cast operator provides a way of forcing the result of an expression to a specific data type. A cast specifies a data type inside parentheses followed by an expression. The expression is first evaluated and then the result is converted (cast) to the specified type. The form of the cast is as follows.

```
(type-name) expression
```

Consider the following example.

```
double tan(), value;  
int    i;  
i = 1;  
value = tan(i);
```

The expression `tan(i)` will not work properly since the tangent function requires an argument of type `double`. However, a cast will allow the integer `i` to be used as an argument to the tangent function.

```
value = tan((double) i);
```

5.7 Comma Operator

The comma operator allows multiple expressions to be used where only a single expression would normally be allowed. An expression list may be formed by separating individual expressions by commas. The expressions in the list are evaluated left to right. The last expression evaluated then becomes the result of the expression list, the results of the previous expressions being discarded. The comma operator groups left to right.

Example:

```
for (i=0, j=0; i<MAXI && j<MAXJ; i++, j++);
```

The `for` statement requires three expressions separated by the semicolon (`;`). In this example, the first and third expressions utilize the comma operator to turn a single expression into two expressions.

5.8 Structure Member Operator

The structure member operator is used to access a member of a structure variable. A period placed between the variable name and a member name of a structure accesses a particular member of the structure.

Example:

declaration of structure:

```
struct {
    char  first_member;
    int   second_member;
    float third_member;
} variable_name;
```

Accessing members of the structure:

```
variable_name.first_member
variable_name.second_member
variable_name.third_member
```

5.9 Structure Pointer Operator

The structure pointer operator is used to access a member of a structure pointer variable. The pointer operator is formed with the minus sign followed by the greater than sign. The `->` symbol placed between the pointer variable name and a member name of a structure accesses a particular member of the structure.

Example:

declaration of structure:

```
struct {
    char  first_member;
    int   second_member;
    float third_member;
} *variable_name;
```

Accessing members of the structure:

```
variable_name->first_member
variable_name->second_member
variable_name->third_member
```

5.10 Conditional Expression

The conditional expression allows the logical value of one expression to determine which of two other expressions will be evaluated. The ternary operator, `?:`, is used to define a conditional expression. The conditional expression has the following form.

`expr1 ? expr2 : expr3`

The above form represents a single expression that is actually composed of three expressions. An expression of this form is evaluated as follows.

- (1) `expr1` is evaluated and determined to be either non-zero (true) or zero (false)
- (2) If `expr1` is non-zero, then `expr2` is evaluated and becomes the result of the conditional expression.
- (3) If `expr1` is zero, then `expr3` is evaluated and becomes the result of the conditional expression.

A conditional expression is very similar to an if-else statement combination.

```
if (expr1)
    expr2;
else
    expr3;
```

The difference is that the conditional expression may be used any place that an expression is legal and it has a result just like any other expression. The result of the conditional expression is the result of either `expr2` or `expr3`. These expressions may yield results of different types. For example, `expr2` might yield a result of type `int` while `expr3` yields a result of type `double`. If the resulting types are different, the final result will be converted to the type having the greatest precision. The normal type conversion rules are applied in such a case. For example, if `expr2` was evaluated as `int`, it would then be converted to `double` because `expr3` is `double`. The conditional expressions group right to left.

Examples:

```
smallest = a < b ? a : b;

printf("This employee is %s.\n",
       salary > 30000 ? "rich" : "poor");
```

5.11 Constant Expressions

A constant expression is an expression that involves only constants. A constant expression is evaluated at compile time and may be used wherever a constant is appropriate.

Examples:

```
#define SIX 6

SIX * 2
2 + 3
sizeof(long)
'A'
"This is a string constant."
```

5.12 Sample Expressions

The following expressions on the right are the parenthesized equivalents to the expressions on the left. They show how the order of evaluation is effected by operator precedence and grouping.

expression	equivalent
<code>*ptr++</code>	<code>*(ptr++)</code>
<code>alpha * - 2</code>	<code>alpha * (- 2)</code>
<code>!x == 0</code>	<code>(!x) == 0</code>
<code>a = - b + c * d - e</code>	<code>a = ((- b) + (c * d)) - e</code>
<code>a = b + c % d - e</code>	<code>a = (b + (c % d)) - e</code>
<code>x = y == z</code>	<code>x = (y == z)</code>
<code>x *= y = z = q</code>	<code>x *= (y = (z = q))</code>
<code>x = y = z = r</code>	<code>x = (y = (z = r))</code>
<code>s = - t + q * r / c</code>	<code>s = ((- t) + ((q * r) / c))</code>
<code>z = a % b * c + d</code>	<code>z = (((a % b) * c) + d)</code>
<code>a = b <= c d != e</code>	<code>a = ((b <= c) (d != e))</code>
<code>a = ! b c && d</code>	<code>a = ((!b) (c && d))</code>
<code>q %= - a++ / c</code>	<code>q %= (((-a++))/c)</code>
<code>-a[j]++</code>	<code>-((a[j])++)</code>
<code>f = g / h ^ ~ i</code>	<code>f = ((g / h) ^ (~i))</code>
<code>f = ~ g & h i ^ j</code>	<code>f = (((~g) & h) (i ^ j))</code>
<code>c *= d /= e += f -= g</code>	<code>c *= (d /= (e += (f -= g)))</code>
<code>q *= x - y ? ++d : e <<= 3</code>	<code>q *= (((x - y) ? (++d) : e) <<= 3)</code>
<code>f = x += 3 , x * y</code>	<code>f = ((x += 3), (x * y))</code>
<code>*str.mem</code>	<code>*(str.mem)</code>

Chapter 6

Functions

C programs are composed of a set of one or more functions. The functions are used to split large programs into smaller, easier to handle pieces. Each function can perform one piece or task. The functions of a program may be defined in any order and may also reside in several different files. Each file may be compiled separately.

Every program must always contain one and only one function named main. Program execution always begins with the main function. The main function may then call other functions which in turn may call still other functions. When a function is called, program execution is transferred to the first statement in the called function. The function terminates after executing a return statement or the last statement in the function. When the called function terminates, program execution returns to the calling function. Execution in the calling function resumes with the evaluation of the expression in which the call was made. The value returned by the function is used as an operand in the expression. In many cases, the function call is the only operand of an expression. In such cases, the returned value is simply discarded.

Data communication between functions is accomplished through shared global (extern) variables and/or through an argument list. The argument list of a function defines variables that are used to contain values passed from other functions.

6.1 Function Definition

Functions

A function definition has the form:

```
data_type function_name(arg1, arg2, ... argn)
/* declarations for arguments arg1...argn */
{
    /* function_body */
}
```

where:

`data_type` is the data type of the value returned by the function.

`function_name` is the name by which the function is called.

`function_body` is the compound statement containing variable declarations and statements

6.1.1 Function Names

A function name is an identifier, so any legal identifier may be used to name a function. The name chosen must not conflict with any other function names or external variable names. Function names may be upper case, lower case, or mixed case. The accepted convention is to make them lower case, the same as variable names. By default, the compiler converts all function names to upper case in the object code output. However, a compiler option is provided to disable this conversion.

6.1.2 Function Types

The data type of a function defines the type of value that the function returns. The return value of a function may be one of the basic data types (char, int, short, long, unsigned, float, double) or a pointer. A function cannot return a composite data type (eg. structure). If the function type is not specified, then it is assumed that the function returns an int (integer). The return statement is used to define the value that a function returns. The value is converted to the type of the function before being returned. If a function does not contain a return statement, the value returned is undefined.

There is a special data type that is used only in function definitions. This is the type void. The void data type specifies that the function does not return a value. A function of type void is equivalent to a procedure in Pascal. The void data type is needed when calling procedures written in Pascal. Before calling a Pascal procedure, declare it as a function of type void. A void function declaration is also required when calling some of the special library functions.

6.1.3 Function Arguments

If no arguments are needed by a function, the function name should be followed by an empty argument list, (). Otherwise, the argument list must specify a name for each argument. All the arguments of a function should be declared immediately following the argument list. Any argument that is not declared is considered to be of type int (integer). The argument declarations must not specify a storage class.

Any basic data type (or a pointer to any type) may be used to declare function arguments. Structures and arrays must be passed using pointers. However, due to the automatic type conversion rules of C, it is best to avoid using some data types. The data types that should not be used for function arguments are char, short, and float. The reason is that these data types are automatically converted up when used in expressions. The char and short data types are converted to int while the float data type is converted to double. These conversions take place in all expressions, including the arguments of a function call. Therefore, the three data types mentioned should not be used to declare function arguments.

note: A compiler option is provided to turn off automatic type conversion. The option applies only to the arguments in a function call. By using the option, it is possible to declare arguments of type char, short, or float.

6.1.4 Function Body

The outermost left and right braces ({}) form a compound statement that encloses the body of a function. The function body may contain declarations as well as statements. All declarations must precede the first statement. The declarations may be for any of the following purposes.

1. Declare local variables (storage class auto or static)
2. Declare global variables (storage class extern)
3. Declare the functions that are called

Local variable declarations create variables that are private to the function. They have a storage class of either auto or static. Auto is the default storage class and int is the default data type.

Local variable declarations:

```
char   alpha;           /* alpha is auto char   */
auto   number;          /* number is auto int   */
auto   float price;     /* price is auto float  */
static count;           /* count is static int  */
static double sum;      /* sum is static double */
```

Global variable declarations do not create variables. They simply allow the function to access a variable that is defined externally (outside of the function). The extern storage class must be specified and the default data type is int.

Global variable declarations:

```
extern char alpha;      /* alpha is extern char */
extern number;          /* number is extern int  */
extern float price;     /* price is extern float */
extern int count;       /* count is extern int   */
extern double sum;      /* sum is extern double  */
```

Function declarations are used to declare the names and types of other functions that are called. It is not necessary to declare a called function that returns the type int. However, it is necessary to declare all called functions that return a type other than int. The compiler assumes all functions to be of type int unless told otherwise. The sole purpose of a function declaration is to tell the compiler the type of value returned by the function. The arguments of the function should not be specified in the declaration. The empty argument list, (), is enough to tell the compiler that the identifier is a function and not a variable.

Function declarations:

```
char   nextbyte();      /* nextbyte returns char */
int    nextword();      /* nextword returns int  */
float  getreal();       /* getreal returns float */
double sin();           /* sin returns double    */
```

The executable statements follow the last declaration in the function. The statements perform the work of the function. Some of the statements might be calls to other functions. The function is terminated when a return statement is executed or when the last statement in the function is executed. A return statement defines the value that the function returns. This value is converted to the type of the function before being returned. If the function does not terminate with a return statement, the returned value is undefined.

6.2 Nested Blocks

A compound statement is often referred to as a block. A function may consist of many blocks and these blocks may be nested. A nested block refers to a block that is enclosed by another block. The outermost compound statement forms a block that encloses the entire function body. Many other blocks may be nested within the function body.

Each block in a function may contain declarations. The only restriction is that all declarations must precede the first statement in the block. Normally, a function will contain declarations only at the beginning of the outermost block. These declarations are visible to all the statements in the function. Declarations that occur at the beginning of a nested block are visible only to the statements inside it.

Declarations inside a nested block are local to the block. They are not visible to outer level blocks. However, the declarations in an outer level block are visible to the nested block. That is, unless the nested block declares a variable with the same name as an outer level variable. If this occurs, the outer level variable is no longer visible to the statements inside the nested block.

```
sample()
{
    int i, j, k;
    i = j = k = 1;
    {
        int i, j, k;
        i = j = k = 2;
        printf("inner level block\n");
        printf("  i=%d, j=%d, k=%d\n", i, j, k);
    }
    printf("outer level block\n");
    printf("  i=%d, j=%d, k=%d\n", i, j, k);
}
```

6.3 The Main Function

Program execution always begins in the function named main. The main function is normally used without arguments. However, there are two arguments that may be used if desired. By convention, the two arguments to main are called argc and argv. Argc and Argv are used to access the command line arguments that are typed when a program is executed. The command line

arguments can be used to pass information into a program. `Argc` is an integer containing the argument count. This is the number of arguments entered on the command line, each argument being separated by one or more blanks. `Argv` is an array containing pointers to each of the arguments entered on the command line. `Argc` then defines the number of elements in `argv` that point to command line arguments. The following example program prints out the command line arguments that are typed when the program is executed.

```
main(argc, argv)
int  argc;      /* number of command line arguments */
char *argv[];  /* pointers to arguments */
{
    int  i;
    printf("argc = %d\n", argc);
    for (i=0; i<argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);
}
```

6.4 Static Functions

The static storage class may be applied to a function definition. The effect of declaring a function to be static is that it becomes invisible to functions in other source files. A static function is typically used when creating a library of functions to be used by several different programs. Such a library might contain functions that are used only by the other functions in the library. For example, one function might simply perform a small task for a larger function. Programs might never need to call the function that performs the small task. Defining this function to be static hides it from the programs that use the library.

```
static clip(x, y)
double *x, *y;
{
    if (*x > 100) *x = 100;
    if (*y > 100) *y = 100;
}

printxy(x, y)
double x, y;
{
    clip(&x, &y);
    printf("x = %f, y = %f\n", x, y);
}
```

6.5 Calling Functions

A function call has the form:

```
function_name(arg1, arg2, ... argn)
```

The named function is called and the arguments arg1 ... argn are passed to the function. If the function requires no arguments, the function name must be followed by an empty argument list, ().

A function call may appear in any C expression. The returned value of the function is used as an operand in the evaluation of the expression. A function call may also appear by itself. In other words, it is the only operand of the expression. In this case, the returned value is simply discarded. A call to a function of type void must appear by itself since no value is returned.

6.6 Function Pointers

A function name that is not followed by an argument list does not result in a function call. Rather, the result is a pointer to the named function. A variable may be declared as a pointer to a function and then assigned to point to a particular function.

Declaration of function pointer variables:

```
int    (*get_byte)();    /*pointer to function returning int*/
double (*trig_function)(); /*pointer to function returning double*/
```

In the previous declarations, the variable name is preceded by the * symbol and followed by an empty argument list, (). The * indicates that the variable is a pointer while the () indicates that it is a pointer to a function. The type specifies the data type of the functions return value. The parentheses around the variable name are necessary. Without them, a function would be declared, not a variable. For example,

```
int    *get_byte();
```

declares a function named get_byte that returns a pointer to an integer.

A function that is referenced as a pointer (no argument list) must be

declared. Otherwise, the compiler will not know that it is a function and will consider it to be an undeclared variable.

Assigning values to function pointer variables:

```
int    getc();          /* declare getc function    */
double sin();          /* declare sin function    */
get_byte = getc;       /* get_byte points to getc */
trig_function = sin;   /* trig_function points to sin */
```

Through the use of the contents of operator (*), a pointer variable that points to a function may be used to call the function. The contents of the pointer variable is the location of the function.

Calling functions through pointer variables:

```
(*get_byte)(stdin);   /* call the getc function */
(*trig_function)(1.0); /* call the sin function  */
```

The function call through a pointer variable looks just like the declaration of the pointer variable. The parentheses around the variable name are necessary.

6.7 Recursion

Each time a function is called, memory is reserved to store its auto (local) variables and then execution begins. The function has been activated. The function remains active until it terminates by executing either a return statement or the last statement in the function. The memory reserved for storing auto variables is then released. The function has been deactivated. The term recursion refers to a programming technique that involves having more than one activation of a function at the same time. There are two types of recursion, direct and indirect.

Direct recursion occurs when a function calls itself. The first call to the function creates the first activation, reserving space for the auto variables. The function begins executing and then encounters a call to itself. This causes a second activation of the function, again reserving space for the auto variables. Since the first activation of the function has not terminated, there are now two activations of the function. Each activation has its own area of memory reserved for the auto variables. This process can continue, creating a new activation of the function each time the call to itself is executed. At some point however, the function must cease calling itself. Otherwise, there would be an infinite loop that would not terminate until all available memory was used by the auto variables. Therefore, the call to itself must be conditional. When a certain condition exists, the function must finish execution and terminate (deactivate) rather

than call itself again. When the function terminates, control is returned to the previous activation. This process then continues until all the activations have terminated and control is returned to the original caller. The following example demonstrates direct recursion. The program computes factorials by calling a recursive function. (eg. $5! = 5 * 4 * 3 * 2 * 1 = 120$)

Example:

```
main()    /* program to compute n! (n factorial) */
{
    int n;
    float factorial();
    printf("<<< Program to compute n! >>>\n\
        Enter an integer number: ");
    scanf("d%", &n);
    printf("%d! = %f", n, factorial(n));
}

float factorial(n) /* recursive function to compute n! */
int n;
{
    if (n == 1)
        return (1);
    else
        return n * factorial(n - 1);
}
```

When the factorial function is called with the argument n , it calls itself $n-1$ times. Each time it calls itself, it reduces the value of the argument n by 1. When the value is equal to 1, it returns rather than calling itself again. The table below shows the value of the argument n and the returned value for each function activation assuming that the original value of n is 5.

activation	value of n	returned value
1	5	120
2	4	24
3	3	6
4	2	2
5	1	1

The other type of recursion is indirect. Indirect recursion occurs when a function is activated through a series of function calls that it began. For example, function a calls function b, function b calls function c, then function c calls function a. Function a has two activations, with activations of function b and function c in between.

Chapter 7

Pointers and Arrays

7.1 Pointers

A pointer is a value corresponding to the address of an object in memory. The words pointer and address are often used interchangeably. Both refer to a location in memory. A pointer may point to an object of any data type. However, all pointers require the same amount of storage, the amount required to store a machine address. A pointer variable is declared using the * operator. The * operator placed in front of a variable name declares the variable as a pointer.

Example Pointer Declarations:

```
char    c, *ptr_to_char; /* ptr_to_char points to a character */
short   s, *ptr_to_short; /* ptr_to_short points to a short   */
int     i, *ptr_to_int; /* ptr_to_int points to an int     */
long    l, *ptr_to_long; /* ptr_to_long points to a long    */
float   f, *ptr_to_float; /* ptr_to_float points to a float  */
double  d, *ptr_to_double; /* ptr_to_double points to a double */
```

A pointer variable is used to store the address of an object in memory. The address of operator (&) is used to obtain the address of a variable (object). Assuming the previous declarations, the following statements assign addresses to the pointer variables.

```
ptr_to_char   = &c; /* address of variable c */
ptr_to_short  = &s; /* address of variable s */
ptr_to_int    = &i; /* address of variable i */
ptr_to_long   = &l; /* address of variable l */
ptr_to_float  = &f; /* address of variable f */
ptr_to_double = &d; /* address of variable d */
```

The contents of operator (*) requires an operand that corresponds to the address of an object in memory. The result of the operation is the value of the object stored at that address. Assuming the previous assignments, the values of the variables c, s, i, l, f, and d may be obtained using the pointer variabl

```

*ptr_to_char;    /* value of c */
*ptr_to_short;   /* value of s */
*ptr_to_int;     /* value of i */
*ptr_to_long;    /* value of l */
*ptr_to_float;   /* value of f */
*ptr_to_double; /* value of d */

```

A variable may even be declared as a pointer to a pointer. If more than one * operator precedes a variable name, the variable is declared as a pointer to a pointer. The following declaration declares three variables, an int, a pointer to an int, and a pointer to a pointer to an int.

```

int    i;
int    *ptr_to_int;
int    **ptr_to_ptr_to_int;

```

The following statements assign values to each of the variables.

```

i = 1;
ptr_to_int = &i;
ptr_to_ptr_to_int = &ptr_to_int;

```

Assuming the previous assignments, all of the following expressions access the value of the variable i, in this case 1.

```

i;                /* result = 1 */
*ptr_to_int;      /* result = 1 */
**ptr_to_ptr_to_int; /* result = 1 */

```

The expression `**ptr_to_ptr_to_int` is evaluated as `*(*ptr_to_ptr_to_int)`. The result of the expression `*ptr_to_ptr_to_int` is the value of `ptr_to_int`, which is the address of `i`. The result of the expression `*(*ptr_to_ptr_to_int)` then is the value of `i`.

7.2 Arrays

An array is a composite data type that is composed of a fixed number of data elements which are all of the same data type. Any data type may be used to declare an array. The elements of an array may even be declared as pointers. An array can have any number of dimensions. The simplest type of array is the single dimension array. The following declarations declare single dimension arrays of various data types.

```

char    s[20];    /* 20 element array of characters */
int     i[11];   /* 11 element array of integers   */
double  *d[32];  /* 32 element array of pointers to
                  double precision floating point */

```

The elements of an array are accessed by specifying a subscript after the array variable name. The first element of an array has a subscript of 0. The last element of the array has a subscript of $n-1$, where n is the number of elements in the array. The elements of the arrays in the previous declaration are accessed as follows.

```

s[0], s[1], ... s[19]
i[0], i[1], ... i[10]
d[0], d[1], ... d[31]

```

When an array name is referenced without a subscript, the result is a pointer to the beginning of the array. This is equivalent to the address of the first element in the array. For example, `s` is equivalent to `&s[0]`, `i` is equivalent to `&i[0]`, and `d` is equivalent to `&d[0]`.

The C string is actually an array of characters. The element following the last valid character in a string must contain the NULL character to indicate the end of the string.

There is no limit to the number of dimensions in an array. The following declarations declare two dimension arrays.

```

int  table[10][10]; /* 100 element array of integers */
char names[90][20]; /* 1800 element array of characters */

```

The variable `table` is an array of 10 elements, each element being an array of 10 integers. The variable `names` is an array of 90 elements, each element being an array of 20 characters.

A multi-dimensioned array may be accessed in various ways. If the array name is used without any subscripts, the result is a pointer to the beginning of the array. Using the previous declarations, `table[0]` results in a pointer to the beginning of the first 10 element array of integers in `table` while `names[0]` results in a pointer to the first 20 element array of characters in `names`. The reference `table[0][0]` accesses the first integer in the first array of 10 integers in `table` while the reference `names[0][0]` accesses the first character in the first array of 20 characters in `names`.

With two-dimension arrays, the first subscript corresponds to the row and the second to the column at which an element is stored. The elements of an array are stored row major. This means that starting at the beginning of the array, the first subscript of the array varies slowest when accessing elements in the order stored. The following table illustrates how the `table` array would be stored in memory.

```

table[0][0] table[0][1] table[0][2] ... table[0][9]
table[1][0] table[1][1] table[1][2] ... table[1][9]
      .
      .
      .
table[8][0] table[8][1] table[8][2] ... table[8][9]
table[9][0] table[9][1] table[9][2] ... table[9][9]

```

7.3 Using Pointers with Arrays

A pointer is a value that points to the location at which an object is stored in memory. When an array is referenced without subscripts, the result is a pointer to an object that happens to be an entire array.

The elements of an array may be accessed using a pointer rather than an array subscript. The following example declares an array of integers and a pointer to an integer.

```
int *ptr, i[20];
```

Then the following assignment statement assigns the variable `ptr` the address of the beginning of the array `i`. Notice that the address of (`&`) operator is not used because the result of an array name without subscripts is a pointer to the beginning of the array.

```
ptr = i;
```

The first character element of array `i` can then be accessed using the pointer variable. The expression `*ptr` would result in the value of the element `i[0]`. Addition or subtraction may be used with the pointer variable to cause it to point to another element in the array. The expression `ptr++` would increment the pointer variable so that it would then point to the element `i[1]`. The expression `*ptr` would then result in the value of the integer stored in the second element of the array `i`.

Notice that when a value is added (or subtracted) with a pointer, the pointer actually changes by the value times the size of the object to which the pointer points. If two pointers point to elements in the same array, subtracting the pointers will result in the number of elements between the two pointers, regardless of the type of the elements in the array.

A function that has an array as an argument can declare the array in either of two ways. The argument may be declared as an array or as a pointer. How the argument is declared determines how the elements of the array must be accessed. The following function has two arguments that are arrays. One is declared as an array variable and the other as a pointer variable. Notice

that the array declaration specifies an empty subscript. The size of the array is not effected by the declaration since arrays are passed as pointers. The declarations are essentially equivalent. The only difference is the way in which the elements of the arrays are referenced. The function assumes that the last valid character in each array is followed by an element having the NULL value (binary 0).

```

sample(array1, array2)
char array1[], *array2;
{
    int i, count;
    i = count = 0;

    while (array1[i++]) count++;
    printf("%d characters in array1\n", count);

    count = 0;
    while (*array2++) count++;
    printf("%d characters in array2\n", count);
}

```

7.4 Array Initialization

The elements of an array may be initialized when the array is declared. The following is the form of an array declaration that includes initializers.

```
data-type name[n] = {value-1, value-2, ... value-n};
```

The values listed between the braces are assigned to the array elements starting with the first element in the array. The following declaration initializes the 5 elements of an array of 5 integers.

```
int i[5] = {0, 1, 2, 3, 4};
```

The previous declaration is equivalent to the following.

```
int i[5];
i[0] = 0; i[1] = 1; i[2] = 2; i[3] = 3; i[4] = 4;
```

When initializers are used in an array declaration, the subscript may be omitted and the compiler will create an array with the number of elements specified in the initializer list. The following declaration is equivalent to the previous one.

```
int i[] = {0, 1, 2, 3, 4};
```

The number of values in an initializer list may be less but not greater than the number of elements in the array. If the number of values listed is less than the size of the array, the compiler will initialize the remaining elements to 0. The following declaration initializes the first 8 elements in the array of 10 floating point numbers. The compiler initializes the last two elements of the array to 0.

```
float number[10] = {0.0, 1.0, 2.0, 3.0,
                   4.0, 5.0, 6.0, 7.0 };
```

A character array may be initialized in two ways. Either the individual character values may be listed or the character values may be specified as a string constant. If specified as a string constant, the braces are not necessary. The following two declarations are equivalent.

```
char    s[] = {'A', 'E', 'I', 'O', 'U', '\0'};
char    s[] = "AEIOU";
```

Notice that the previous declarations create an array of 6 characters. The compiler automatically appends the NULL character ('\0') to the end of a string constant. Therefore, the declaration that specifies the individual character values must include the NULL character for the two declarations to be equivalent.

An array of pointers can be initialized. The following declaration initializes an array of pointers to characters. Each element of the array day is a pointer to a string.

```
char    *day[] = {"sunday",
                 "monday",
                 "tuesday",
                 "wednesday",
                 "thursday",
                 "friday",
                 "saturday"};
```

Multi-dimensioned arrays may also be initialized. The following declaration initializes a two dimension array. When multiple dimensioned arrays are initialized, the subscripts must be specified.

```
char names[5][20] = {"George Jones    ";
                    "Sam Smith      ";
                    "Shirley Cartwright ";
                    "Debra Johnson   ";
                    "Johnny Abercrombie "};
```

Chapter 8

Structures and Unions

8.1 Structures

A structure provides a means of defining a single variable that is actually composed of several variables. A structure is a group of one or more variables called members that are referenced using a single variable name as a prefix. Unlike arrays, the elements (members) of a structure do not have to be the same data type. There are various ways to create a structure. A structure template defines the form of a structure but does not allocate memory. A structure variable must be declared using the structure template to allocate memory.

8.1.1 Defining Structures

A structure template has the following form.

```
struct struct-tag {
    data-type member-name;
    .
    .
    .
    data-type member-name;
};
```

The struct-tag following struct is an identifier called a structure tag. The structure tag is not required. The tag simply provides a name for the structure template. If the structure template is tagged, then the name may be used for later definitions and declarations. The structure tag serves a purpose similar to a type name defined using typedef.

A structure is also a type. It is valid to use

```
struct {
    int value1;
    int value2;
} a, b;
```

just as it is valid to use

```
int a, b;
```

The structure above declares `a` and `b` as structure variables with two integer members. Since there is no tag, other variables cannot be declared without redefining the structure. Tagging the structure template provides a means of declaring other variables without redefining the structure.

Example:

```
struct date {          /* Define a structure template to */
    int month;         /* contain a date                */
    int day;
    int year;
};
```

Variables of the above structure type may be declared as follows.

```
struct date birthdate; /* birthdate is a structure
                       variable of type date */
struct date graduation; /* graduation is a structure
                        variable of type date */
```

The previous declarations declare two structure variables, `birthdate` and `graduation`. Both of these variables have three integer members. If no structure variables are given after a structure definition (as is the case for `date`), then no storage is reserved. When the declarations are made for `birthdate` and `graduation`, storage is reserved for the two structure variables. Each of these variables require enough space to store three integers. The following defines the structure named `date` and also declares two variables.

```
struct date {
    int month;
    int day;
    int year;
} birthdate, graduation;
```

The previous declaration reserves space for the two structure variables, `birthdate` and `graduation`. The structure template named `date` may still be used for later definitions and declarations of structure variables.

A structure template may have another structure template as a member.

Example:

```
struct date {
    int month, day, year;
};

struct address {
    char street[30];
    char city[20], state[3];
    long zipcode;
};

struct name {
    char last[20], first[20], middle_initial;
};

struct pupil {
    struct name fullname;
    unsigned idnumber;
    long ssn;
    struct date birthdate;
    struct address school_addr, home_addr;
    int classcode;
    int hours_completed, degree_code;
    int major_code, minor_code;
    float overall_gpa, major_gpa;
} student;
```

Notice the following things in the preceding example.

- (1) Structure definitions used as members in the structure named pupil are defined prior to the definition of pupil.
- (2) struct name fullname; declares a member variable called fullname that is a structure of type name.
- (3) struct address school_addr, home_addr; declares two member variables that are structures of type address.
- (4) The data types of the members of the structures need not be the same.

8.1.2 Referencing Structures

Members of a structure variable are referenced as follows.

```
variable-name.member-name
```

The member operator connects a member name to a structure variable name. For example, members of the previously defined structure variable `student` are referenced as follows.

```
student.fullname  student.idnumber  etc.
```

The reference `student.fullname` references a whole structure of type `name`. The individual members of the structure are referenced by appending the appropriate member name.

```
student.fullname.last  student.fullname.first  etc.
```

There is no limit to the level of nesting of structures. The whole structure is referenced by the structure variable name. The members of the structure are then referenced by appending a member name. If a member is a structure, its members are referenced by appending another member name, etc.

The members of structures are assigned values in the same way as other variables using the assignment operator. The type of a member determines the type of value that may be assigned to it.

Whole structures may also be assigned. When assigning whole structures, the structures must be of the same type.

8.1.3 Structure Initialization

Initialization of structure variables is very similar to initialization of arrays. The structure variable name is followed by a list of initializers enclosed in braces.

```
struct date birthdate = {3, 9, 55};
```

The above declaration declares the variable `birthdate` as a structure of type `date` and assigns values to its three members. The following is equivalent.

```

struct date birthdate;

birthdate.month = 3;
birthdate.day = 9;
birthdate.year = 55;

```

The following example illustrates how the previously defined structure variable named student may be initialized.

```

struct pupil student = {
    {"Public", "John", 'Q'}, /* struct fullname    */
    12345, 999999999,        /* idnumber, ssn    */
    {7, 28, 1955},          /* struct birthdate */
    {"Dorm Room 2612C",     /* struct school_addr */
     "Anytown", "TX", 75075},
    {"100 Main Street South", /* struct home_addr  */
     "Big City", "TX", 75000},
    3, 32, 1,               /* classcode,
                               hours_completed,
                               degree_code    */
    60, 73,                 /* major_code, minor_code */
    3.0, 3.2};              /* overall_gpa, major_gpa */

```

8.1.4 Pointers to Structures

A pointer to a structure is declared as

```

struct struct-tag *variable-name;

```

The * preceding the variable name declares the variable as a pointer to a structure of the type specified by struct-tag.

Example:

```

struct address *addrptr; /* pointer to a structure of
                          type address */
struct name *nameptr;   /* pointer to a structure of
                          type name */

```

A pointer must be assigned to point to a specific structure of the type specified by the declaration. The following example assigns the pointer variable addrptr the address of the structure variable permanent_addr.

```

struct address *addrptr, permanent_addr;
addrptr = &permanent_addr;

```

The structure pointer operator -> is used to access a member of a structure

using a pointer variable.

Example:

```

struct name {                /* define structure    */
    char last[20];
    char first[20];
    char middle[20];
} *nptr, fullname;          /* declare variables */

nptr = &fullname;           /* assign nptr the address
                             of fullname */
nptr->last = "Jones";        /* assign values to the
                             members of fullname */
(*nptr).first = "John";

nptr->middle = "Quincy";

```

Since `nptr` is a pointer to `fullname`, either method shown to assign values to the structure members is valid. The `()` around `(*nptr)` are required because the structure member operator has higher precedence than the contents of operator. Without the `()`, the expression would be equivalent to `*(nptr.last)`.

8.1.5 Arrays of Structures

The following definition declares an array of structures.

```

struct name {
    char last[25];
    char first[15];
    char middle;
} fullname[10];

```

The variable `fullname` is a 10 element array, each element being a structure of type `name`. The reference `fullname[0]` accesses the complete structure in the first element of the array. The reference `fullname[0].last` accesses the member named `last` in the first element of the array. The reference `fullname[0].last[0]` accesses the first character in the member named `last` in the first element of the array.

8.1.6 Bit Fields

A member of a structure may be defined as a bit field. A bit field defines the number of bits of storage that the member requires. A bit field is specified by following the member name by a colon (:) and then the number of bits required for storage. This is only allowed for members of type `int` or `unsigned`. All bit field values are treated as unsigned integers in expressions. Bit fields allow several members of a structure to be stored in a single integer. For example, if the size of an integer is 16 bits, then 16 one bit members may be stored in a single integer. Or perhaps you might store 4 four bit members in a single integer.

The maximum size of a single bit field is the number of bits in an `int`. Consecutive members of a structure that are declared as bit fields are stored in consecutive bits of an integer. The starting bit may be either the least or most significant bit of the integer, depending on the machine. A bit field can not span across an `int` boundary. Therefore, a bit field that requires more bits than the number of bits remaining in an `int` will be stored starting at the next `int` boundary.

The minimum size of a bit field is 1. However, the `sizeof(int)` is the minimum amount of storage allocated for each consecutively declared bit field members.

When declaring structure members using bit fields, you may simply specify a bit field (`:n`) without a data type or member name. This is used for padding to force the next bit field member to start at a particular bit within the `int`. A field width of 0 may be specified to force the next bit field member to start at the next `int` boundary. Padding is not necessary if the next structure member is not a bit field. Structure members that are not bit fields always begin at the next `int` boundary if they follow a bit field member.

The following example illustrates a structure that has members declared using bit fields.

```
struct student_record {
    char    name[40];
    unsigned out_of_state:1;
    unsigned tb_test:1;
    unsigned physical:1;
    unsigned govt_aid:1;
    unsigned housing:1;
    unsigned parking:1;
    unsigned registered:1;
    unsigned category:3;
           :0;
    char    last_initial;
};
```

The `student_record` structure makes use of bit fields to save space. The members from `out_of_state` to `registered` are all 1 bit fields that are used to represent a true/false state. True can be represented by the bit being set (1) and false by the bit being clear (0). Only 7 bits are required to store these 7 members. The last bit field member, `category`, uses 3 bits of storage. A field of 3 bits can represent a number between 0 and 7. In all, the bit field members require only 10 bits of storage. This is less than the `sizeof(int)`. The `:0` is used to force the next member to the next int boundary. However, it is not required since the next member is not a bit field. The `sizeof(struct student_record)` is equal to `41 + sizeof(int)` since the bit field members all fit in a single integer. Assigning values to bit field members is identical to any other assignment using structure members. You should be careful not to assign a value outside the range of the bit field. For example, a one bit field can only have two values, 0 or 1.

8.2 Unions

A union has the same form as a structure. However, all the members of a union share a common area of memory. This allows a single area of memory to be used for storing several different data types. The data types of the members of a union may all be different. The amount of memory allocated to a union variable is the size of the largest member in the union.

```
union union-tag {
    data-type member-name;
    .
    .
    .
    data-type member-name;
} variable-name;
```

As with structures, the `union-tag` and `variable-name` are optional.

The union provides a way to access the same area of memory in different ways. Each of the union members overlap one another in the same area of memory. Different parts of this area of memory may be accessed by referencing the appropriate union member. The following example illustrates a union with two members sharing the same area of memory. One of the members is an unsigned integer. The other is a structure that is composed of 16 single bit members. The two members are exactly the same size provided that the `sizeof(int)` is 16. The structure member then overlaps the integer member exactly. The structure member can then be used to access the individual bits of the integer member. The example shows how the union, together with bit fields, may be used to set the bits of the format flag that is used with the `ftoa` library function. To use bit fields in this manner, you must know a little about the hardware of the computer. On some computers, integers are stored with the most significant bit first. On others, the least significant bit is first. Therefore, a program like this example is machine dependent.

On some machines, the order of the bit field members should be reversed.

```

union format {
    unsigned all;
    struct {
        unsigned bit0:1;
        unsigned bit1:1;
        unsigned bit2:1;
        unsigned bit3:1;
        unsigned bit4:1;
        unsigned bit5:1;
        unsigned bit6:1;
        unsigned bit7:1;
        unsigned bit8:1;
        unsigned bit9:1;
        unsigned bit10:1;
        unsigned bit11:1;
        unsigned bit12:1;
        unsigned bit13:1;
        unsigned bit14:1;
        unsigned bit15:1;
    } part;
};

main()
{
    int i;
    char s[80];
    union format flag;
    flag.all=0; /* turn all bits off */
    flag.part.bit4=1; /* turn on leading $ sign */
    flag.part.bit6=1; /* turn on include commas */
    ftoa(1000., s, flag.all, 10, 2);
    puts(s);
}

```


Chapter 9

Statements

9.1 Simple and Compound Statements

An expression followed by a semicolon (;) is a statement. The semicolon is a statement terminator and not a statement separator as in other languages such as Pascal. Braces {} are used to group several statements into a compound statement. The braces cause the enclosed statements to be treated as one statement rather than several individual statements. A compound statement may be used anywhere a simple statement may be used. No semicolon appears after a right brace since braces are not statements.

9.2 Conditional Statements

9.2.1 if

An if statement has the form:

```
if (expression) statement;
```

If the expression is non-zero (true), then the statement is executed. If the expression is zero (false), then the statement is not executed.

Example:

```
toupper(c)
int c;
{
    if (c >= 'a' || c <= 'z') c -= 32;
    return c;
}
```

9.2.2 else

An else statement has the form:

```
else statement;
```

The else statement may only be used in conjunction with an if statement. An else statement is matched with the closest unmatched if statement. The if statement must precede the else statement. If the expression of the matching if statement is zero (false), then the else statement is executed. If the expression of the matching if statement is non-zero (true), then the else statement is not executed.

Example:

```
#define TRUE 1
#define FALSE 0

isspace(c)
int c;
{
    if (c == '\t' || c == '\n' || c == ' ') return TRUE;
    else return FALSE;
}
```

A common programming technique for selecting one of several statements to execute is to use a sequence of if and else statements. The following is the form of such a sequence.

```
if (expression)
    statement;
else if (expression)
    statement;
else if (expression)
    statement;
    .
    .
    .
else
    statement;
```

The previous sequence of if-else statements will select and execute only one of the statements. The first expression that results in a non-zero (true) value will cause the immediately following statement to be executed. The remaining statements in the sequence are then skipped. If none of the expressions result in a true value, then the last else statement is executed. The following example is an illustration of an if-else sequence.

Example:

```
int alpha, digit, space, other;

count(c);
int c;
{
    if (isalpha(c))
        alpha++;
    else if (isdigit(c))
        digit++;
    else if (isspace(c))
        space++;
    else
        other++;
}
```

9.2.3 switch

The switch statement has the form:

```
switch (expression) {

    case const1: statement;
                statement;

    case const2: statement;
                statement;
                .
                .
                .

    case constn: statement;
                statement;

    default: statement;
            statement;
}
```

The switch statement is used to select a statement or sequence of statements to execute based on the value of an expression. The expression is evaluated and the value compared to the constant values following each case keyword. These values must be integer or character constants or constant expressions. Execution begins at the statement following the constant that matches the value of the expression. If none of the constant values match the value of the expression, execution begins at the statement following the default keyword. The default case is optional. If it is omitted and no match is found, then none of the statements inside the switch statement are executed.

All of the statements inside the switch statement that follow the matching constant value are executed. In other words, all the statements following the matched case to the end of the switch statement are executed, not just the statements following the matched case. Since this is normally not desired, the last statement in each individual case is the break statement. The break statement causes the remaining statements inside the switch to be skipped.

Example:

```
int digit, space, other;

count(c)
int c;
{
    switch (c) {
        case '0' :
        case '1' :
        case '2' :
        case '3' :
        case '4' :
        case '5' :
        case '6' :
        case '7' :
        case '8' :
        case '9' : digit++;
                    break;
        case '\t':
        case '\n':
        case ' ' : space++;
                    break;
        default: other++;
    }
}
```

Notice that multiple case constants may be specified without any following statements. Execution begins at the first statement following the matched constant.

9.3 Looping Statements

9.3.1 while

The while statement has the form:

```
while (expression) statement;
```

The while statement uses a conditional expression to determine whether or not to execute a statement. First the expression is evaluated. While the expression is non-zero (true) the statement is repetitively executed. When the expression is zero (false), the while statement is terminated.

Example:

```
#include "stdio"
main()
{
    int c, count = 0;
    while ((c = getchar()) != EOF) {
        putchar(c);
        count++;
    }
    printf("%d characters copied", count);
}
```

9.3.2 do-while

A do-while statement has the form:

```
do statement while (expression);
```

The do-while statement uses a conditional expression to determine whether or not to continue executing a statement. First the statement is executed and then the expression is evaluated. The statement is repetitively executed while the expression is non-zero (true). When the expression becomes zero (false), the do-while statement is terminated.

Example:

```
#include "stdio"
main()
{
    int c;
    do {
        c = getchar();
        if (isdigit(c)) count++;
    }
    while (c != EOF);
    printf("%d alphabetic characters", count);
}
```

9.3.3 for

A for statement has the form:

```
for (expr1; expr2; expr3) statement;
```

The for statement is a general looping statement that is normally used to execute a statement a specific number of times. First expr1 is evaluated. This expression is evaluated only once. This is usually an assignment statement that initializes a counter variable. Then expr2 is evaluated. This is the conditional expression. If expr2 is non-zero (true), the statement is executed and then expr3 is evaluated. Expr3 is usually an expression that increments the counter variable. While expr2 is non-zero (true), the statement and expr3 are repetitively executed. When expr2 is zero (false), the for statement is terminated.

A for statement is equivalent to the following statements.

```
expr1;
while (expr2) {
    statement;
    expr3;
}
```

Any of the 3 expressions may be omitted. The semicolon must remain as a place holder for the missing expression. For example, for(;;) is perfectly legal. Since there is no conditional expression, this is an infinite loop.

Example:

```
main()
{
    int i;
    char  digit[10];
    for (i = 0; i < 10; i++) {
        digit[i] = i + '0';
        printf("digit[%d] = %c\n", i, digit[i]);
    }
}
```

9.4 break

The break statement causes an immediate exit from a for, while, do-while, or switch statement. If one of these statements is inside another, the break statement exits the immediately enclosing statement. For example, a break

statement inside a switch statement that is inside a for statement causes only the switch statement to be exited.

In the following example, the break statement is used to exit both a switch statement and a for statement.

Example:

```
#include "stdio"
main()
{
    int c, vowels = 0, lines = 0;
    for(;;) {
        c = getchar();
        switch (toupper(c)) {
            case 'A' :
            case 'E' :
            case 'I' :
            case 'O' :
            case 'U' : vowels++;
                       break;
            case '\n': lines++;
        }
        if (c == EOF) break;
    }
    printf("%d vowels in %d lines", vowels, lines);
}
```

9.5 continue

The continue statement may be used inside a for, while, or do-while statement. The continue statement causes the next iteration of the enclosing loop to begin. Continue is related to break in that both cause execution of the current loop iteration to terminate. The difference lies in the fact that continue begins the next iteration of the enclosing loop while break exits the enclosing loop. Continue causes execution to jump to expr3 of a for statement and to the conditional expressions in while and do-while statements.

The following function reads an integer from the file fp. The continue statement is used to skip over leading whitespace characters.

Example:

```
#include "stdio"
getint(fp)
FILE *fp;
{
    int negative = 0, value = 0;
    for(;;)
        c = getc(fp);
        if (isspace(c)) continue;
        if (c == '-')
            ++negative;
        c = getc(fp);
    }
    if (!isdigit(c)) return EOF;
    while (isdigit(c))
        value = value * 10 + c - '0';
        c = getc(fp);
    }
    ungetc(c, fp);
    if (negative) return -value;
    else return value;
}
}
```

9.6 goto and labels

The goto statement has the form:

```
goto identifier;
```

A labeled statement has the form:

```
identifier: statement;
```

The goto statement is used to branch to a labeled statement. A statement is labeled by placing an identifier followed by a colon in front of the statement. The goto statement then references the identifier.

Example:

```
main()
{
    goto end;
    puts("This is not printed");
end: puts("goto labeled statement");
}
```

9.7 return

The return statement has the form:

```
return expression;
```

The return statement is used to exit a function. The result of the expression is the value returned by the function. The expression is optional. If no expression is given, the value returned is undefined.

Example:

```
tolower(c)
int c;
{
    if (c >= 'A' && c <= 'Z') return c + 32;
    else return c;
}
```

9.8 null

The null statement has the form:

```
;
```

The null statement may be used anywhere a statement is legal. It is useful with loops when there are no statements inside the loop. The following loop reads characters until a non-whitespace character is read.

```
while (isspace(c = getchar()));
```


Chapter 10

Input and Output

When a program begins execution, three files are automatically opened:

```
stdin  is opened for reading  (input device: keyboard)
stdout is opened for writing   (output device: screen)
stderr is opened for writing   (output device: screen)
```

note: These files are not automatically opened if the main function is named `_main` rather than `main`.

The `stdin` and `stdout` files may be remapped to other devices (such as disk files) when a program is executed. The System Implementation Manual explains how to remap `stdin` and `stdout`. `stderr` is always mapped to the screen.

`stdin`, `stdout`, and `stderr` are file pointers defined in the standard header file, `stdio`. Some of the standard input and output functions use these file pointers implicitly. Others require that a file pointer be specified as one of the arguments to the function. Some of the functions return the values `EOF` or `NULL`. Both of these values are defined in the standard header file. A return value of `EOF` or `NULL` indicates that either the end of file or an error was detected.

10.1 Opening and Closing Files

All input and output is performed through an opened file. The `fopen` function is used to open a file. Once a file is no longer needed, it should be closed. The `fclose` function is used to close an opened file.

10.1.1 fopen

Format:

```
fp = fopen(name, mode);

FILE    *fp;    /* file pointer */
char    *name;  /* file name   */
char    *mode;  /* access mode */
```

Description:

The fopen function opens a file. The name specifies the name of the file. The System Implementation Manual describes device names that may also be used. The mode specifies how the file will be accessed. The three access modes are:

```
"r" to open the file for reading
"w" to open the file for writing
"a" to open the file for appending
```

A file must exist to be opened for reading. Otherwise, the function returns an error indication. A file need not exist to be opened for writing or appending. If it doesn't, a file will be created. Opening a file for writing erases an existing file. Opening a file for appending causes subsequent output to be appended to the end of an existing file.

Returns:

```
fp = pointer if successful
NULL      if unsuccessful
```

Example:

```
main()
{
    FILE    *fp;
    fp = fopen("database", "w");
}
```

10.1.2 fclose

Format:

```
status = fclose(fp);

int      status; /* return status */
FILE     *fp     /* file pointer  */
```

Description:

The `fclose` function closes the file pointed to by the file pointer `fp`. All open files are closed automatically when a program terminates normally or when the `exit` function is called. However, the `fclose` function allows you to close a single file at any given time. There are two reasons for closing a file explicitly. First, the contents of an output file that is not properly closed may be lost. If a program abnormally terminates, the opened files are not automatically closed and the contents of any opened output files will probably be lost. Second, there is a limit to the number of files that may be open at a given time. This limit is defined in the standard header file as `MAXFILES`. By closing a file when it is no longer needed, the file pointer becomes available for use by another file.

Returns:

```
status = 0 if successful
        -1 if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    int status;
    FILE *fp;
    fp = fopen("database", "w");
    status = fclose(fp);
}
```

10.2 Character I/O

There are several functions that perform input or output a single character at a time.

10.2.1 getchar

Format:

```
c = getchar();  
  
int    c;
```

Description:

The `getchar` function returns the next character from the file pointed to by `stdin`.

Returns:

```
c = character if successful  
    EOF      if unsuccessful
```

Example:

```
#include "stdio"  
main()  
{  
    int    c;  
    while ((c = getchar()) != EOF);  
}
```

10.2.2 putchar

Format:

```
    status = putchar(c);

    int     status; /* return status */
    int     c;      /* character    */
```

Description:

The `putchar` function outputs the character `c` to the file pointed to by `stdout`.

Returns:

```
    status = c    if successful
            EOF   if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    int     c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

10.2.3 getc

Format:

```
    c = getc(fp);

    int     c;
    FILE    *fp;
```

Description:

The `getc` function returns the next character from the file pointed to by `fp`.

Returns:

```
c = character if successful
EOF      if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    int    c;
    FILE   *input;
    input = fopen("infile", "r");
    if (input != NULL)
        while ((c = getc(input)) != EOF);
}
```

10.2.4 putc

Format:

```
status = putc(c, fp);

int    status; /* return status */
int    c;      /* character      */
FILE   *fp;    /* file pointer  */
```

Description:

The putc function outputs the character c to the file pointed to by fp.

Returns:

```
status = c    if successful
EOF          if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    int    c;
    FILE   *input, *output;
    input  = fopen("infile", "r");
    output = fopen("outfile", "w");
    if (input != NULL && output != NULL)
        while ((c = getc(input)) != EOF)
            putc(c, output);
}
```

10.2.5 ungetc

Format:

```
status = ungetc(c, fp);

int    status;
int    c;
FILE   *fp;
```

Description:

The `ungetc` function returns the character `c` to the input file pointed to by `fp`. The next character received from this file will then be the returned character.

Returns:

```
status = character if successful
        EOF        if unsuccessful
```

Example:

```
#include "stdio"
getnumber(fp);
FILE   *fp;
{
    int    c;
    int    number = 0;
    while ((c = getc(fp)) != EOF && isspace(c));
    if (!isdigit(c)) return EOF;
    else {
        while(isdigit(c)) {
            number = 10 * number + (c - 48);
            c = getc(fp)
        }
        ungetc(c, fp);
        return number;
    }
}
```

10.3 String I/O

The following functions input or output a string of characters. The input functions read an entire line of characters from the input file. Input is terminated when the newline character ('\n') is encountered. The output functions write a string of characters to the output file. The end of a string is marked by the NULL character ('\0').

10.3.1 gets

Format:

```
    status = gets(s);

    int     status;    /* return status */
    char    *s;       /* string buffer */
```

Description:

The gets function reads the next line from the file pointed to by stdin into the string buffer pointed to by s. Characters are read into the string buffer until the newline character is encountered. The newline character is then discarded and a NULL character is appended to mark the end of the string.

Returns:

```
    status = s    if successful
             NULL if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    char    s[81];
    while (gets(s) != NULL);
}
```

10.3.2 puts

Format:

```
    status = puts(s);

    int     status;    /* return status */
    char    *s;       /* string buffer */
```

Description:

The puts function writes the string of characters pointed to by s to the file pointed to by stdout. Characters are written from the string buffer to the file until the NULL character is encountered. The NULL character is then discarded and the newline character is written to start a new line.

Returns:

```
    status = s    if successful
            EOF  if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    char    s[81];
    while (gets(s) != NULL) puts(s);
}
```

10.3.3 fgets

Format:

```
    status = fgets(s, n, fp);

    int     status;    /* return status */
    char    *s;       /* string buffer */
    int     n;        /* size of buffer */
    FILE    *fp;      /* file pointer */
```

Description:

The `fgets` function reads the next line from the file pointed to by `fp` into the string buffer pointed to by `s`. Characters are read into the string buffer until the newline character is encountered or until `n-1` characters have been read, whichever occurs first. The argument `n` prevents the function from reading characters after the buffer becomes full. The newline character is passed through to the string buffer. A NULL character is then appended to mark the end of the string.

Returns:

```
status = s    if successful
        NULL if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    #define BUFSIZE 81
    char    s[BUFSIZE];
    FILE    *input;
    input = fopen("infile", "r");
    if (input != NULL)
        while (fgets(s, BUFSIZE, input) != NULL);
}
```

10.3.4 fputs

Format:

```
status = fputs(s, fp);

int    status;    /* return status */
char   *s;        /* string buffer */
FILE   *fp;       /* file pointer  */
```

Description:

The `fputs` function writes the string of characters pointed to by `s` to the file pointed to by `fp`. Characters are written from the string buffer to the file until the NULL character is encountered. The NULL character is discarded.

Returns:

```
status = s    if successful
        EOF   if unsuccessful
```

Example:

```
#include "stdio"
main()
{
    #define BUFSIZE 81
    char    s[BUFSIZE];
    FILE    *input;
    input = fopen("infile", "r");
    output = fopen("outfile", "w");
    if (input != NULL && output != NULL)
        while (fgets(s, BUFSIZE, input) != NULL)
            fputs(s, output);
}
```

10.4 Formatted I/O

The following functions allow formatted input or output of characters, strings, and numbers. Each requires a format string as an argument. Following the format string is a list of other arguments. For input functions, the arguments are addresses of the variables that store the data read from the input file. For output functions, the arguments are variables (or constants) containing the data that is written to the output file.

The format string describes the format of the input or output data. A format string may contain ordinary text and/or format conversion specifications. A format conversion specification is required for each argument that follows the format string. The first conversion specification in the format string is matched with the first argument following the format string. The second conversion specification is matched with the second argument and so on.

The format conversion specifies the type of data being input or output. It specifies whether the data is a single character, a string of characters, or one of the numeric data types. Numeric data types must be converted from string to binary format during input and from binary to string format during output.

10.4.1 Input Format Strings

An input format string may contain any of the following:

Whitespace	Blanks, tabs, and newlines are ignored.
Characters	All other characters except % are expected to match the next non-whitespace character in the input. This is used to skip over characters in the input.
Conversions	A conversion specification causes the next input field to be read. The specification begins with the character %, followed by an optional assignment suppression character *, followed by an optional maximum field width, followed by the conversion character.

An input field is defined by the type of data it contains. For all data types except character, an input field begins with the next non-whitespace character in the input. For strings, the field extends to the next whitespace character. For numeric data, the field extends to either the next whitespace character or the next invalid numeric character, whichever occurs first. For character, the field is simply the next character.

When an input field is read, the value is normally assigned to one of the arguments (variables). However, if the conversion specifies the assignment suppression character *, the assignment is not made. The assignment suppression character tells the input function to discard the input field rather than assign it to a variable. A conversion specification that includes the assignment suppression character is not matched with an argument following the format string. Therefore, an argument should not be provided for conversions that specify assignment suppression.

The optional field width specifies the maximum number of characters in an input field. The field width should be specified as a positive integer. When a field width is specified, the next input field is defined as the smallest of its actual width or the specified maximum width.

The conversion character specifies the type of the input field. Some of the conversion characters may be preceded by the letter l.

- d or ld A decimal integer is expected in the input. For d, the matching argument should be a pointer to an int. For ld, the matching argument should be a pointer to a long. All leading whitespace characters are skipped. All consecutive decimal digits are then read, converted to an integer value, and then assigned.
- o or lo An octal integer (with or without a leading 0) is expected in the input. For o, the matching argument should be a pointer to an int. For lo, the matching argument should be a pointer to a long. All leading whitespace characters are skipped. All consecutive octal digits are then read, converted to an integer value, and then assigned.
- x or lx A hexadecimal integer (with or without a leading 0x) is expected in the input. For x, the matching argument should be a pointer to an int. For lx, the matching argument should be a pointer to a long. All leading whitespace is skipped. All consecutive hexadecimal digits are then read, converted to an integer value, and then assigned.
- h A short decimal integer is expected in the input. The matching argument should be a pointer to a short. All leading whitespace is skipped. All consecutive decimal digits are then read, converted to an integer value, and then assigned.
- c A single character is expected in the input. The matching argument should be a pointer to a char. The next character is read and then assigned.
- s A string of characters is expected in the input. The matching argument should be a pointer to an array of char. An array name without subscripts is treated as a pointer to the first element in the array. The array must be large enough to hold the input string plus a terminating NULL character ('\0'). All leading whitespace is skipped. All consecutive non-whitespace characters are then read and assigned.

- f or lf A floating point number is expected in the input. For f, the matching argument should be a pointer to a float. For lf, the matching argument should be a pointer to a double. All leading whitespace is skipped. All consecutive characters comprising a legal floating point number are then read, converted to a floating point value, and then assigned to the floating point variable. A legal floating point number consists of an integer part, a fraction part, and an exponent part. Either the integer part or fraction part may be missing but not both. The exponent part may be missing.
- e or le This is equivalent to f and lf.
- y A dynamic string is expected in the input. The matching argument should be a pointer to a dynamic string variable. All leading whitespace is skipped. All consecutive non-whitespace characters are then read and assigned to the dynamic string variable. A dynamic string is an extended data type that is defined in the standard header file as STRING. There are several functions described in the System Implementation Manual that use dynamic strings.

The following examples assume that the input stream is:

123 456 789

Conversion Spec	number of characters read	values input
-----	-----	-----
%c	1	'1'
%d	3	123
%ld	3	123
%x	3	291
%o	3	83
%u	3	123
%h	3	123
%f	3	123.
%lf	3	123.
%s	3	"123"
%d%f%s	14	123, 456., "789"
.*%c	4	' '
%s.*%d	7	"123", 456
%2s%c	3	"12", '3'
%2s%d%2s	6	"12", 3, "45"

10.4.2 Output Format Strings

An output format string may contain any of the following:

Characters	All characters except % are simply written to the output.
Conversions	A conversion specification causes the next argument to be converted and output. The conversion specification begins with the character %, followed by an optional left adjustment character -, followed by an optional number specifying the minimum field width, followed by an optional number specifying the precision (must be preceded by a period), followed by the conversion character.

The output field width is determined by the data type of the argument. A character has a field width of one. A string has a field width equal to the number of characters in the string. A number has a field width equal to the number of digits required to represent the number.

The optional left adjustment character -, causes the argument to be left justified in its field. This has no effect unless a minimum field width is specified.

The optional field width specifies the minimum field width for an argument. If the field width required to represent the argument is greater than the minimum field width, then the minimum field width has no effect. However, if the minimum field width is greater, then the the output field is padded on the left (or right if left justification is specified) to fill out the minimum width. The padding character is normally a blank. However, if the minimum field width is specified with a leading 0, the padding character is a zero.

The optional precision is used only for string or floating point arguments. The precision is specified as a positive integer number preceded by a period. For string arguments, the precision specifies the maximum number of characters to output from the string. For floating point arguments, the precision specifies the number of digits to output to the right of the decimal point. The default precision for floating point arguments is 6.

The conversion character specifies the type of the output data. Some of the conversion characters may be preceded by the letter l.

d or ld	The output format is signed decimal. For d, the matching argument should be a short or int value. For ld, the matching argument should be a long value.
o or lo	The output format is unsigned octal. For o, the matching argument should be a short or int value. For lo, the matching argument should be a long value.
x or lx	The output format is unsigned hexadecimal. For x, the matching argument should be a short or int value. For lx, the matching argument should be a long value.
u	The output format is unsigned decimal. The matching argument should be an unsigned int value.
c	The output format is a single character. The matching argument should be a char value.
s	The output format is a sequence of characters. The matching argument should be a pointer to a string of characters. The string must be terminated by a NULL character ('\0').
e	The output format is exponential. Exponential format is used to represent floating point numbers in the form [-]m.nnnnnnEsxx, where the minus sign is printed if the number is negative, the number of n's is defined by the precision, and E represents the exponent (s is the sign of the exponent and xx is the integer value of the exponent). The matching argument should be a float or double value.

- f The output format is fixed point. Fixed point format is used to represent floating point numbers in the form `[-]mmm.nnnnnn`, where the minus sign is printed if the number is negative (blank if positive) and the number of n's is defined by the precision. The matching argument should be a float or double value.
- g The output format is the shorter of the e or f formats. In either case, the non-significant trailing zeros are not printed. The matching argument should be a float or double.
- y The output format is a dynamic string. The matching argument should be a pointer to a dynamic string. A dynamic string has the type `STRING`, defined in the standard header file.

Any other characters that follow a % character in the output format string are simply printed. For example, the sequence `%%` in the format string would cause the single % character to be printed.

In the following examples, the fields are bounded by `:` to show exactly what is being output. The conversion specification is given, then the field as it would look after being output.

Output Formatting for Strings

The string is "I am an old C dog." which has 18 characters.

conversion spec	output field
(1) %15s	:I am an old C dog.:
(2) %-15s	:I am an old C dog.:
(3) %25s	: I am an old C dog.:
(4) %-25s	:I am an old C dog. :
(5) %25.15s	: I am an old C d:
(6) %-25.15s	:I am an old C d :
(7) %.12s	:I am an old :

Explanation

- (1) Outputs the entire string since the minimum field width is less than the length of the string.
- (2) Outputs the entire string since the minimum field width is less than the length of the string.
- (3) Right justifies and pads unused field width with blanks.
- (4) Left justifies and pads unused field width with blanks.
- (5) Right justifies a maximum of 15 characters in a field width of 25 characters.
- (6) Left justifies a maximum of 15 characters in a field width of 25 characters.
- (7) Outputs a maximum of 12 characters.

Output Formatting for Numbers

Assume the following variable declarations:

```
short  s = 123;
int    i = 12345;
long   l = 1234567;
float  f = 12.34560;
double d = -123456.789;
```

conversion spec	variable	output field
(1) %010d	s	:0000000123:
(2) %-10d	i	:12345 :
(3) %10ld	l	: 1234567:
(4) %5f	f	:12.345600:
(5) %5.2f	d	:-123456.79:
(6) %10e	f	:1.234560E+01:
(7) %10.2e	d	: -1.23E+05:
(8) %10.5g	f	: 12.3456:
(9) %5.2g	d	:-1.23E+05:

Explanation:

- (1) The short integer is right justified. The padding character is a zero since the minimum field width has a leading zero.
- (2) The integer is left justified.
- (3) The long integer is right justified, padded with blanks.
- (4) The float requires a 9 character field, including the default precision of 6 digits to the right of the decimal point.
- (5) The double requires a 10 character field. Notice that floating point numbers are rounded before being truncated.
- (6) The float requires a 12 character field when printed in exponential format. The default precision is 6.
- (7) The float is right justified and printed in exponential format with a precision of 2.
- (8) The float takes fewer characters in fixed format. Notice that trailing non-significant zeros are not printed.
- (9) The float takes fewer characters in exponential format.

10.4.3 scanf

Format:

```

status = scanf(fs, arg1, arg2, ... argn);

int    status; /* return status */
char   *fs;    /* pointer to format string */

```

Description:

The scanf function reads values from the file pointed to by stdin and assigns them to the arguments, arg1 through argn. The fs argument points to the format string that tells scanf the data type of each of the other arguments passed to it. Arg1 through argn must be variable addresses since scanf stores the values read in the memory locations pointed to by arg1 through argn. The format string must contain a format conversion specification for each of the arguments, arg1 through argn. Any other text in the format string simply causes matching text in the file to be skipped.

Returns:

```
status = number of values assigned if successful
        number of values assigned if error
        EOF if end of file
```

Example:

```
main()
{
    char    c;
    short   s;
    int     i;
    unsigned u;
    long    l;
    float   f;
    double  d;
    char    string[81];

    printf("Input Using Scanf:\n\n");
    scanf("%c, %h, %d, %u, %ld, %f, %lf, %s",
          &c, &s, &i, &u, &l, &f, &d, string);
}
```

10.4.4 printf

Format:

```
status = printf(fs, arg1, arg2, ... argn);

int    status; /* return status */
char   *fs;    /* pointer to format string */
```

Description:

The `printf` function writes `arg1` through `argn` to the file pointed to by `stdout`. The `fs` argument points to the format string that tells `printf` the data type of each of the other arguments passed to it. The format string must contain a format conversion specification for each of the arguments, `arg1` through `argn`. Any other text in the format string is simply written to `stdout`.

Returns:

```
status = 0    if successful
          EOF if unsuccessful
```

Example:

```
main()
{
    char    c = 'a';
    short   s = 127;
    int     i = 32767;
    unsigned u = 65535;
    long    l = 100000;
    float   f = 1.23456;
    double  d = 1.23456789;
    char    string[] = "TheEnd.";

    printf("Output Using Printf:\n\n");
    printf("%c %d %d %u %l %f %f %s",
           c, s, i, u, l, f, d, string);
}
```

10.4.5 `fscanf`

Format:

```
status = fscanf(fp, fs, arg1, arg2, ... argn);

int     status; /* return status */
FILE    *fp;    /* file pointer */
char    *fs;    /* pointer to format string */
```

Description:

The `fscanf` function reads values from the file pointed to by `fp` and assigns them to the arguments, `arg1` through `argn`. The `fs` argument points to the format string that tells `fscanf` the data type of each of the other arguments passed to it. `arg1` through `argn` must be variable addresses since `fscanf` stores the values read in the memory locations pointed to by `arg1` through `argn`. The format string must contain a format conversion specification for each of the arguments, `arg1` through `argn`. Any other text in the format string simply causes matching text in the file to be skipped.

Returns:

```
status = number of values assigned if successful
        number of values assigned if error
        EOF if end of file
```

Example:

```
main()
{
    char    c;
    short   s;
    int     i;
    unsigned u;
    long    l;
    float   f;
    double  d;
    char    string[81];

    printf("Input Using Fscanf:\n\n");
    fscanf(stdin, "%c, %h, %d, %u, %ld, %f, %lf, %s",
           &c, &s, &i, &u, &l, &f, &d, string);
}
```

10.4.6 fprintf

Format:

```
status = fprintf(fp, fs, arg1, arg2, ... argn);

int     status; /* return status */
FILE    *fd;    /* file pointer */
char    *fs;    /* pointer to format string */
```

Description:

The `fprintf` function writes `arg1` through `argn` to the file pointed to by `fp`. The `fs` argument points to the format string that tells `fprintf` the data type of each of the other arguments passed to it. The format string must contain a format conversion specification for each of the arguments, `arg1` through `argn`. Any other text in the format string is simply written to `fp`.

Returns:

```
status = 0    if successful
          EOF if unsuccessful
```

Example:

```
main()
{
    char    c = 'a';
    short   s = 127;
    int     i = 32767;
    unsigned u = 65535;
    long    l = 100000;
    float   f = 1.23456;
    double  d = 1.23456789;
    char    string[] = "TheEnd.";

    fprintf(stdout, "Output Using Fprintf:\n\n");
    fprintf(stdout, "%c %d %d %u %l %f %f %s",
               c, s, i, u, l, f, d, string);
}
```

10.4.7 `sscanf`

Format:

```
status = sscanf(s, fs, arg1, arg2, ... argn);

int     status; /* return status */
char    *s;     /* pointer to input string */
char    *fs;    /* pointer to format string */
```

Description:

The `sscanf` function reads values from the string pointed to by `s` and assigns them to the arguments, `arg1` through `argn`. The `fs` argument points to the format string that tells `sscanf` the data type of each of the other arguments passed to it. `Arg1` through `argn` must be variable addresses since `sscanf` stores the values read in the memory locations pointed to by `arg1` through `argn`. The format string must contain a format conversion specification for each of the arguments, `arg1` through `argn`. Any other text in the format string simply causes matching text in the input string to be skipped.

Returns:

status = number of values assigned if successful
 number of values assigned if error

Example:

```
main()
{
    char    c;
    short   s;
    int     i;
    unsigned u;
    long    l;
    float   f;
    double  d;
    char    string[81];
    char    s[] =
        "a 127 32767 65535 100000 1.23456 1.23456789 TheEnd";

    printf("Input Using Sscanf:\n\n");
    sscanf(s, "%c, %h, %d, %u, %ld, %f, %lf, %s",
           &c, &s, &i, &u, &l, &f, &d, string);
}
```

10.4.8 sprintf

Format:

```

    status = sprintf(s, fs, arg1, arg2, ... argn);

    int     status; /* return status */
    char    *s;     /* pointer to output string */
    char    *fs;    /* pointer to format string */

```

Description:

The sprintf function writes arg1 through argn to the string pointed to by s. The fs argument points to the format string that tells fprintf the data type of each of the other arguments passed to it. The format string must contain a format conversion specification for each of the arguments, arg1 through argn. Any other text in the format string is simply written to s.

Returns:

```

    status = 0   if successful
             EOF if unsuccessful

```

Example:

```

main()
{
    char    c = 'a';
    short   s = 127;
    int     i = 32767;
    unsigned u = 65535;
    long    l = 100000;
    float   f = 1.23456;
    double  d = 1.23456789012345;
    char    string[] = "The End.";
    char    s[81];

    printf("Output Using Sprintf:\n\n");
    sprintf(s, "%c %d %d %u %l %f %f %s\0",
            c, s, i, u, l, f, d, string);
    printf("%s", s);
}

```


Chapter 11

Standard Functions

11.1 Character Functions

A set of standard functions are used to determine the type of a character. A character may be alphabetic, digit, whitespace, lower case, or upper case. Two functions convert case. One converts upper case to lower case while the other converts lower case to upper case.

11.1.1 isalpha

Format:

```
result = isalpha(c);

int    result; /* true or false result */
int    c;      /* character */
```

Description:

The isalpha function compares the argument c with the alphabetic characters 'A' through 'Z' and 'a' through 'z'. The argument c may also be of type char since it is automatically converted to int.

Returns:

```
result = non-zero (true)  if c is alphabetic
         zero         (false) if c is not alphabetic
```

Example:

```
#include "stdio"
main()
{
    int c;
    for (c=0; c<128; c++)
        if (isalpha(c)) putchar(c);
}
```

11.1.2 isdigit

Format:

```
result = isdigit(c);

int    result; /* true or false result */
int    c;      /* character */
```

Description:

The `isdigit` function compares the argument `c` with the digit characters '0' through '9'. The argument `c` may also be of type `char` since it is automatically converted to `int`.

Returns:

```
result = non-zero (true)  if c is digit
         zero         (false) if c is not digit
```

Example:

```
#include "stdio"
main()
{
    int c;
    for (c=0; c<128; c++)
        if (isdigit(c)) putchar(c);
}
```

11.1.3 isspace

Format:

```
result = isspace(c);

int    result; /* true or false result */
int    c;      /* character */
```

Description:

The `isspace` function compares the argument `c` with the whitespace characters ' ' (blank), '\t' (tab), and '\n' (newline). The argument `c` may also be of type `char` since it is automatically converted to `int`.

Returns:

```
result = non-zero (true)  if c is whitespace
         zero        (false) if c is not whitespace
```

Example:

```
#include "stdio"
main()
{
    int c;
    for (c=0; c<128; c++)
        if (isspace(c)) putchar(c);
}
```

11.1.4 islower

Format:

```
result = islower(c);

int    result; /* true or false result */
int    c;      /* character */
```

Description:

The `islower` function compares the argument `c` with the lower case alphabetic characters 'a' through 'z'. The argument `c` may also be of type `char` since it is automatically converted to `int`.

Returns:

```
result = non-zero (true)  if c is lower case
         zero          (false) if c is not lower case
```

Example:

```
#include "stdio"
main()
{
    int c;
    for (c=0; c<128; c++)
        if (islower(c)) putchar(c);
}
```

11.1.5 `isupper`

Format:

```
result = isupper(c);

int    result; /* true or false result */
int    c;      /* character */
```

Description:

The `isupper` function compares the argument `c` with the upper case alphabetic characters 'A' through 'Z'. The argument `c` may also be of type `char` since it is automatically converted to `int`.

Returns:

```
result = non-zero (true)  if c is upper case
         zero          (false) if c is not upper case
```

Example:

```
#include "stdio"
main()
{
    int c;
    for (c=0; c<128; c++)
        if (isupper(c)) putchar(c);
}
```

11.1.6 tolower

Format:

```
result = tolower(c);

int    result; /* converted character */
int    c;      /* character */
```

Description:

The tolower function compares the argument *c* with the alphabetic characters 'A' through 'Z'. If *c* is an upper case alphabetic character, the function returns the lower case equivalent. Otherwise the function returns *c* unchanged. The argument *c* may also be of type *char* since it is automatically converted to *int*.

Returns:

```
result = lower case equivalent if c is upper case
        c if c is not upper case
```

Example:

```
#include "stdio"
main()
{
    int c;
    for (c=0; c<128; c++)
        if (isalpha(c)) putchar(tolower(c));
}
```

11.1.7 toupper

Format:

```
result = toupper(c);

int    result; /* converted character */
int    c;      /* character */
```

Description:

The toupper function compares the argument c with the alphabetic characters 'a' through 'z'. If c is a lower case alphabetic character, then the function returns the upper case equivalent. Otherwise the function returns c unchanged. The argument c may also be of type char since it is automatically converted to int.

Returns:

```
result = upper case equivalent if c is lower case
        c if c is not lower case
```

Example:

```
#include "stdio"
main()
{
    int c;
    for (c=0; c<128; c++)
        if (isalpha(c)) putchar(toupper(c));
}
```

11.2 String Functions

There are several standard functions that operate on strings. A string is defined as a sequence of characters terminated by the NULL character ('\0').

11.2.1 strlen

Format:

```
length = strlen(s);

int    length; /* length of the string */
char   *s;     /* pointer to the string */
```

Description:

The `strlen` function determines the length (in bytes) of a string of characters. The argument `s` is a pointer to the beginning of the string. The string must be terminated by the NULL character (`'\0'`). The NULL character is not counted in determining the length.

Returns:

length = number of characters (bytes) in the string

Example:

```
main()
{
    int length;
    char s[] = "0123456789";

    length = strlen(s);
    printf("length of \"0123456789\" is %d", length);
}
```

11.2.2 strcpy

Format:

```
strcpy(dest, source);

char *dest; /* pointer to the destination string */
char *source; /* pointer to the source string */
```

Description:

The `strcpy` function copies the string pointed to by the source argument to the string pointed to by the dest argument. The copying stops when the NULL character is encountered in the source string. The NULL character is then appended to the destination string.

Returns:

none

Example:

```
main()
{
    char dest[81];
    char source[] = "0123456789";

    strcpy(dest, source);
    puts(source);
    puts(dest);
}
```

11.2.3 `strcmp`

Format:

```
result = strcmp(s1, s2);

char    *s1; /* pointer to the first string */
char    *s2; /* pointer to the second string */
```

Description:

The `strcmp` function compares two strings. The string argument `s1` is compared with the string argument `s2`. Starting with the first character in each string, the characters of the two strings are sequentially compared until a character in the first string does not match a character in the second. The value returned by the function is then calculated by subtracting the two non-matching characters. The character in `s2` is subtracted from the character in `s1`. If `s1` is identical to `s2`, then the value returned by the function is 0.

Returns:

```
result = negative if s1 < s2
         0         if s1 = s2
         positive if s1 > s2
```

Example:

```
main()
{
    char s1[81];
    char s2[81];

    puts("*** String Compare ***");
    printf("Enter the first string : ");
    scanf("%s", s1);
    printf("Enter the second string : ");
    scanf("%s", s2);
    if (strcmp(s1,s2) < 0)
        printf("%s < %s", s1, s2);
    else if (strcmp(s1,s2) > 0)
        printf("%s > %s", s1, s2);
    else
        printf("%s = %s", s1, s2);
}
```

11.2.4 strcat

Format:

```
strcat(s1, s2);

char *s1; /* pointer to the first string */
char *s2; /* pointer to the second string */
```

Description:

The `strcat` function concatenates two strings. The string argument `s2` is appended to the end of string argument `s1`. The size of the string `s1` must be large enough to hold both `s1` and `s2` with a NULL byte appended at the end.

Returns:

none

Example:

```
main()
{
    char s1[120];
    char s2[120];

    puts("*** String Concatenate ***");
    printf("Enter the first string : ");
    scanf("%s", s1);
    printf("Enter the second string : ");
    scanf("%s", s2);
    strcat(s1, s2);
    printf("%s", s1);
}
```

11.2.5 strsave

Format:

```
ptr = strsave(s);

char *ptr; /* pointer to a copy of string s */
char *s    /* pointer to string s          */
```

Description:

The strsave function saves a copy of a string in the heap. The string argument *s* is copied into the heap and the returned value is a pointer to where the copy of string *s* is located.

Returns:

```
ptr = pointer to the copy of string s if successful
      NULL if not enough space in the heap to store s
```

Example:

```
#include "stdio"
#define MAX 50
char *strings[MAX];
main()
{
    char s[81];
    int i = 0;
    char *strsave();

    while (scanf("%s", s) != EOF && i < MAX)
        strings[i++] = strsave(s);
    i = 0;
    while (strings[i] != NULL && i < MAX)
        puts(strings[i++]);
}
```

11.3 Dynamic String Functions

A non-standard type of string is the dynamic string. A dynamic string has a structure as defined by `STRING` in the standard header file. Dynamic strings are stored in the heap. Two functions are provided for converting a normal string to a dynamic string and vice versa. The System Implementation Manual describes a library of string functions that operate only on dynamic strings.

11.3.1 stods

Format:

```
ds = stods(s);

STRING *ds; /* pointer to the dynamic string */
char *s; /* pointer to the normal string */
```

Description:

The `stods` function converts a normal string to a dynamic string. The normal string argument `s` is converted to a dynamic string. The function returns a pointer to the dynamic string. Dynamic strings are required by the library of string functions described in the System Implementation Manual. The `scanf` and `printf` functions have been extended to read and write dynamic strings using the conversion specification `%y`.

Returns:

ds = pointer to the dynamic string if successful
NULL if not enough space in heap for the dynamic string

Example:

```
#include "stdio"
main()
{
    char    s[] = "abcdefghijklmnopqrstuvwxyz";
    STRING *ds;
    STRING *stods();

    ds = stods(s);
    printf("dynamic string = %y\n", ds);
}
```

11.3.2 dstos

Format:

```
dstos(ds, s);

STRING *ds; /* pointer to the dynamic string */
char    *s; /* pointer to the normal string */
```

Description:

The dstos function converts the dynamic string pointed to by the argument ds to a normal string. The string is stored at the location pointed to by the argument s.

Returns:

none

Example:

```
#include "stdio"
main()
{
    char    s[81];
    STRING *ds;

    printf("Enter a string: ");
    scanf("%y", ds);
    dstos(ds, s);
    printf("%s", s);
}
```

11.4 Conversion Functions

There are two functions that convert a string to a number. One function converts an integer string to an int. The integer string is a string of digit characters representing a valid integer number. The other function converts a floating point string to a double. The floating point string is a string of characters representing a valid floating point number (See Floating Point Constants in Chapter 1).

There are also two functions that convert a number to a string. One function converts an int to a string of digit characters. The other converts a double to a string of characters representing the floating point value.

11.4.1 atoi

Format:

```
i = atoi(s);

int    i;    /* binary value of integer string */
char   *s;   /* pointer to the integer string */
```

Description:

The atoi function converts a string to its integer equivalent. The argument *s* is a pointer to the string of digits which must represent a value in the range of an int. The function converts the string to a binary integer value.

Returns:

`i` = binary value of the integer string `s`

Example:

```
main()
{
    char    s[] = "12345";
    int     i;

    i = atoi(s);
    printf("%d", i);
}
```

11.4.2 atof

Format:

```
f = atof(s);

double f; /* binary value of floating point string */
char *s; /* pointer to the floating point string */
```

Description:

The `atof` function converts a string to its floating point equivalent. The argument `s` is a pointer to the floating point string which must represent a value in the range of a double. The function converts the string to a binary floating point value.

Returns:

`f` = binary value of the floating point string `s`

Example:

```
main()
{
    char    s[] = "1.2345e2";
    double  f;
    double  atof();

    f = atof(s);
    printf("%f", f);
}
```

11.4.3 itoa

Format:

```
    itoa(i, s);

    int    i;    /* binary value of integer */
    char  *s;    /* pointer to the integer string */
```

Description:

The itoa function converts a binary integer value to its string equivalent. The argument *i* is the binary integer value and the argument *s* is a pointer to the location where the digit string is stored.

Returns:

none

Example:

```
main()
{
    char    s[10];
    int     i = 236;

    itoa(i, s);
    printf("%d %s", i, s);
}
```

11.4.4 ftoa

Format:

```
    ftoa(f, s, flag, left, right);

    double  f;    /* binary value of double */
    char    *s;    /* pointer to the floating point string */
    unsigned flag; /* format flag */
    unsigned left; /* number of digits left of decimal */
    unsigned right; /* number of digits right of decimal */
```

Description:

The `ftoa` function is a non-standard function that converts a binary floating point value to an equivalent string of characters. The argument `f` is a binary floating point value. The argument `s` is a pointer to where the string equivalent is stored. The argument `flag` specifies the format of the string conversion. The argument `left` specifies the number of characters to the left of the decimal point, while the argument `right` specifies the number of characters to the right of the decimal point. The buffer pointed to by argument `s` should be larger than is necessary to store the converted floating point value. The size required is effected by the value of the floating point number and the three arguments: `flag`, `left`, and `right`. To be safe, you should make the buffer size 80. The `ftoa` function is guaranteed not to use more than 80 characters under any conditions.

The value of the `flag` argument may simply be 0. Then the floating point number is converted in fixed point format. The `left` and `right` arguments define the number of characters to the left and right of the decimal point. If the argument `left` is larger than needed, blanks will precede the number. If the argument `right` is larger than is needed, 0's will follow the number.

The `flag` argument provides considerable control over the format of the converted number. The `flag` contains 8 bits that may be turned on or off to control the format of the conversion. Constants can be defined to represent each of the 8 bits. Then the `|` (bitwise or) operator may be used to turn on the bits desired. The following chart defines the function of each of the 8 bits in the format `flag`. A function is turned off if its bit is clear (0) and on if its bit is set (1).

Bit	Function	Example
0	exponential format	1.23e+01 instead of 12.3
1	no trailing 0's	1.23 instead of 1.2300
2	sign follows	1.23- instead of -1.23
3	include sign	+1.23 instead of 1.23
4	leading \$ sign	\$1.23 instead of 1.23
5	leading * sign	**1.23 instead of 1.23
6	include commas	1,234.56 instead of 1234.56
7	change leading + to blank	1.23 instead of +1.23

Returns:

none

Example:

```

#define BIT0 1    /* exponential format      */
#define BIT1 2    /* no trailing 0's      */
#define BIT2 4    /* sign follows         */
#define BIT3 8    /* include sign         */
#define BIT4 16   /* leading $ sign       */
#define BIT5 32   /* leading * sign       */
#define BIT6 64   /* include commas       */
#define BIT7 128  /* change leading + to blank */

main()
{
    char    s[80];
    double  f = -1234.567890;
    unsigned flag = BIT2 | BIT4 | BIT6;
    unsigned left = 10, right = 2;

    ftoa(f, s, flag, left, right);
    puts(s);
}

```

11.5 Dynamic Memory Functions

A section of memory called the heap is available for a program to use as needed for storing variables. The program controls the use of the heap through two standard functions, `calloc` and `cfree`. The `calloc` function allocates memory. The `cfree` function frees memory for other use.

11.5.1 `calloc`

Format:

```

ptr = calloc(number, size);

char *ptr;          /* ptr to allocated block of memory */
unsigned number;    /* number of units to allocate */
unsigned size;      /* size of a unit in bytes */

```

Description:

The `calloc` function allocates (reserves) a block of memory. The size of the block (in bytes) is determined by multiplying the number argument (number of units) by the size argument (size of a unit). The `calloc` function then allocates a block of this size and returns a pointer to the beginning of the block.

Notes: The `sizeof` operator is useful in determining the size of a type or variable. The memory allocated by `calloc` will be cleared (set to zero) if the `/*$ZERO*/` compiler option is used.

Returns:

`ptr` = pointer to the allocated block if successful
NULL (not enough space in heap) if unsuccessful

Example:

```
struct _name {
    char first[15];
    char middle[15];
    char last[15];
};

main()
{
    int status;
    struct _name *name;
    char *calloc();
    name = calloc(1, sizeof(struct _name));
    if (name != NULL) {
        printf("Enter Name (first middle last): ");
        scanf("%s%s%s",
            name->first, name->middle, name->last);
        printf("%s%s%s",
            name->first, name->middle, name->last);
    }
    else printf("<< Out of space in the heap >>");
}
```

11.5.2 cfree

Format:

```
cfree(ptr);  
  
char *ptr; /* pointer to allocated block of memory */
```

Description:

The cfree function frees (releases) a block of memory that was previously allocated by calloc. The ptr argument is a pointer to the beginning of the block of memory. The entire block of memory to which ptr points is released for other use.

Returns:

none

Example:

```
struct _name {  
    char first[15];  
    char middle[15];  
    char last[15];  
};  
  
main()  
{  
    int status;  
    struct _name *name;  
    char *calloc();  
    name = calloc(1, sizeof(struct _name));  
    if (name != NULL) {  
        printf("Enter Name (first middle last): ");  
        scanf("%s%s%s",  
            name->first, name->middle, name->last);  
        printf("%s %s %s",  
            name->first, name->middle, name->last);  
        cfree(name);  
    }  
    else printf("<< Out of space in the heap >>");  
}
```

11.6 Math Functions

The math functions provide the ability to perform scientific calculations. These functions must be declared because they all return double values. All arguments are double values and must be expressed in radians. Degrees may be converted to radians by multiplying by 3.141592654/180.

11.6.1 abs

Format:

```
f1 = abs(f2);

double f1; /* absolute value of f2 */
double f2; /* floating point value */
```

Description:

The abs function returns the absolute value of the floating point argument f2.

Returns:

```
f1 = absolute value of f2
```

Example:

```
main()
{
    double f = -1.0;
    double abs();

    printf("abs(%f) = %f", f, abs(f));
}
```

11.6.2 atan

Format:

```
f1 = atan(f2);

double f1; /* arctangent of f2 */
double f2; /* floating point value */
```

Description:

The atan function returns the arctangent of the floating point argument f2.

Returns:

f1 = absolute value of f2

Example:

```
main()
{
    double f = 1.0;
    double atan();

    printf("atan(%f) = %f", f, atan(f));
}
```

11.6.3 cos

Format:

```
f1 = cos(f2);

double f1; /* cosine of f2      */
double f2; /* floating point value */
```

Description:

The cos function returns the cosine of the floating point argument f2.

Returns:

f1 = cosine of f2

Example:

```
#define PI 3.141592654
main()
{
    double f = PI;
    double cos();

    printf("cos(%f) = %f", f, cos(f));
}
```

11.6.4 exp

Format:

```
f1 = exp(f2);

double f1; /* natural exponential of f2 */
double f2; /* floating point value */
```

Description:

The exp function returns the natural exponential of the argument f2. The exp function uses base e (2.718281828) and raises this value to the f2 power.

Returns:

f1 = natural exponential of f2

Example:

```
main()
{
    double f = 1.0;
    double exp();

    printf("exp(%f) = %f", f, exp(f));
}
```

11.6.5 log

Format:

```
f1 = log(f2);

double f1; /* natural logarithm of f2 */
double f2; /* floating point value */
```

Description:

The log function returns the natural logarithm of the argument f2. The log function uses base e (2.718281828) in taking the logarithm of f2. Only positive values are allowed for argument f2.

Returns:

f1 = natural logarithm of f2

Example:

```
main()
{
    double f = 2.718281828;
    double log();

    printf("log(%f) = %f", f, log(f));
}
```

11.6.6 sin

Format:

```
f1 = sin(f2);

double f1; /* sine of f2          */
double f2; /* floating point value */
```

Description:

The sin function returns the sine of the floating point argument f2.

Returns:

```
f1 = sine of f2
```

Example:

```
#define PI 3.141592654
main()
{
    double f = PI/2;
    double sin();

    printf("sin(%f) = %f", f, sin(f));
}
```

11.6.7 sqr

Format:

```
f1 = sqr(f2);

double f1; /* square of f2          */
double f2; /* floating point value */
```

Description:

The `sqr` function returns the square of the argument `f2`. The `sqr` function simply returns `f2 * f2`.

Returns:

`f1` = square of `f2`

Example:

```
main()
{
    double f = 2.0;
    double sqr();

    printf("sqr(%f) = %f", f, sqr(f));
}
```

11.6.8 `sqrt`

Format:

```
f1 = sqrt(f2);

double f1; /* square root of f2 */
double f2; /* floating point value */
```

Description:

The `sqrt` function returns the square root of the argument `f2`. Only values greater than or equal to 0 are allowed for argument `f2`.

Returns:

`f1` = square root of `f2`

Example:

```
main()
{
    double f = 4.0;
    double sqrt();

    printf("sqrt(%f) = %f", f, sqrt(f));
}
```

11.7 Termination Functions

There are two standard functions that will terminate a program when called. The `exit` function terminates a program after closing all open files. The `_exit` function terminates a program without closing files.

11.7.1 `exit`

Format:

```
    exit(status);

    int    status;    /* termination status */
```

Description:

The `exit` function closes all open files and then terminates the program. If the status argument is 0, the program is terminated normally. If status is -1, the program is aborted. A program that is aborted will terminate an executing batch stream while a normal program termination will not. The status argument may not have any effect at all under some operating systems.

Returns:

None

Example:

```
main()
{
    FILE *fp;
    fp = fopen("test", "w");
    fputs("This is a test", fp);
    exit(0);
    fputs("This is not printed", fp);
}
```

11.7.2 `_exit`

Format:

```
    _exit(status);  
    int      status;    /* termination status */
```

Description:

The `_exit` function terminates the program without closing any files. The contents of any output files that are open when `_exit` is called may be lost. The status argument performs the same action as described for the `exit` function.

Returns:

None

Example:

```
main()  
{  
    FILE *fp;  
    fp = fopen("test", "w");  
    fputs("This is a test", fp);  
    _exit(0);  
    fputs("This is not printed", fp);  
}
```

Chapter 12

Compiler Controls

12.1 Preprocessor Statements

Statements beginning with the # sign are known as preprocessor statements. The term preprocessor is used for historical reasons. Most C compilers make a separate pass through a C program to process just the statements beginning with the # sign, creating an intermediate file that is then compiled. The # sign is usually required to begin in the first column. This compiler however makes only a single pass through the source program and the # sign may begin in any column. The following sections explain the preprocessor statements.

12.1.1 #include

Format:

```
#include "file_name"
```

where file_name is the name of a disk file

Description:

The #include statement is used to include another C source file during a compile. When the compiler encounters a #include statement, it compiles the C source in the named file before continuing. It has the same effect as inserting the source in the file at the location of the #include statement. The file name used with #include must be enclosed by either double quotes "" or angle brackets <>. On some systems, the angle brackets may cause the file to be searched for on a specific disk drive.

Example:

```
#include "stdio"
#include "globals"
main()
{
    #include "locals"
    #include "body"
}
```

12.1.2 #define

Format:

```
#define MACRO_NAME macro_definition

where MACRO_NAME is an identifier
and macro_definition is a text string
```

Description:

The #define statement is used to define a macro. A macro is an identifier and the text that the compiler substitutes when the identifier is later encountered in the program. The identifier corresponds to the name of the macro while the text corresponds to the actual definition of the macro.

The main use of macros is to define program constants. However, a macro can also have arguments. This allows macros to be used as shorthand notations for complex expressions that are used in many different places in a program. An argument list may follow the identifier to provide a means of passing information to the macro definition. The macro definition text would then contain the arguments as part of the text string. The macro is used much like a function call. The identifier is followed by an argument list and the compiler makes the appropriate substitution. The arguments passed to a macro are passed as text.

Example:

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#define MIN(a,b) (((a) < (b)) ? (a) : (b))

main()
{
    float  f1, f2;
    printf("Enter 2 numbers: ");
    scanf("%f%f", &f1, &f2);
    printf("The largest number is %f\n", MAX(f1,f2));
    printf("The smallest number is %f\n", MIN(f1,f2));
}
```

12.1.3 #undef

Format:

```
#undef MACRO_NAME
```

where MACRO_NAME is an identifier

Description:

The #undef statement is used to undefine a macro. An undefined macro is no longer known to the compiler. The #undef statement can be used to free the space used by a macro definition that is no longer needed.

Example:

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#define MIN(a,b) (((a) < (b)) ? (a) : (b))

main()
{
    float  f1, f2;
    printf("Enter 2 numbers: ");
    scanf("%f%f", &f1, &f2);
    printf("The largest number is %f\n", MAX(f1,f2));
    printf("The smallest number is %f\n", MIN(f1,f2));
    #undef MAX
    #undef MIN
}
```

12.1.4 #ifdef

Format:

```
#ifdef MACRO_NAME
    statements
#endif
```

Description:

The `#ifdef` statement is used to conditionally compile selected statements of a program only if the named macro is defined. The conditionally compiled statements are preceded by the `#ifdef` and followed by a `#endif`. The program lines in between are then compiled only if the specified macro is defined. The compiler listing will show a `-` sign preceding a line that is not compiled.

Example:

```
#include "stdio"
#define DEBUG

writeline(s, fp)
char *s;
FILE *fp;
{
    fputs(s, fp);
    putc('\n', fp);
    #ifdef DEBUG
    puts(s);    /* output to screen */
    #endif
}
```

12.1.5 #ifndef**Format:**

```
#ifndef MACRO_NAME
    statements
#endif
```

Description:

The `#ifndef` statement is used to conditionally compile selected statements of a program only if the named macro is not defined. The conditionally compiled statements are preceded by the `#ifndef` and followed by a `#endif`. The program lines in between are then compiled only if the specified macro is not defined. The compiler listing will show a `-` sign preceding a line that is not compiled.

Example:

```
#include "stdio"
#include "constant"

#ifndef MAXBUF
#define MAXBUF 81
#endif

main()
{
    char s[MAXBUF];
    while (gets(s) != NULL) puts(s);
}
```

12.1.6 #if

Format:

```
#if constant-expression
    statements
#endif
```

Description:

The #if statement is used to conditionally compile selected statements of a program only if the constant-expression is non-zero (true). The conditionally compiled statements are preceded by the #if and followed by a #endif. The program lines in between are then compiled only if the expression does not evaluate to 0 (false). The compiler listing will show a - sign preceding a line that is not compiled.

Example:

```
#define DOLLAR 1
#define COMMA 1

main()
{
    int flag;
    char s[20];
    #if DOLLAR && COMMA
        flag = 80;
    #endif
    #if DOLLAR && !COMMA
        flag = 16;
    #endif
    #if !DOLLAR && COMMA
        flag = 64;
    #endif
    #if !DOLLAR && !COMMA
        flag = 0;
    #endif
    ftoa(1000.0, s, flag, 8, 2);
    puts(s);
}
```

12.1.7 #else

Format:

```
#else
    statements
```

Description:

The #else statement may be used with the #ifdef, #ifndef, or #if statements. The #else and its accompanying statements should be placed just prior to the #endif. The #else statements are compiled only if the preceding statements (of the #ifdef, #ifndef, or #if) are not compiled.

Example:

```
#define MOD1 0
#define MOD3 0

#if MOD1 || MOD3
#define SCREEN_WIDTH 64
#define SCREEN_HEIGHT 16
#else
#define SCREEN_WIDTH 80
#define SCREEN_HEIGHT 24
#endif

main()
{
    printf("width = %d\n", SCREEN_WIDTH);
    printf("height = %d", SCREEN_HEIGHT);
}
```

12.1.8 #line

Format:

```
#line constant
```

Description:

The compiler numbers lines sequentially starting at 1 when creating a program listing. The #line statement may be used to set the line number of the next line of the compiler listing. The compiler will then number subsequent lines of the listing beginning with the specified line number.

Example:

```
#include "stdio"
#line 1
main()
{
    /* main starts at line 1 */
}
```

12.2 Compiler Options

Compiler options are switches that control various characteristics of the compiler. A compiler option is specified using a comment. The format is as follows.

```
/*$COMPILER_OPTION*/ or /*$NO COMPILER_OPTION*/
```

All compiler options must be specified in upper case and there must be no spaces between the /* and the \$ sign.

A compiler option may be turned on or off. The option is turned on unless preceded by NO, in which case it is turned off. Except where noted, compiler options may appear anywhere in a program. They may be turned on or off as needed. For each option, there is a default setting. If the default setting is the one desired, then the option need not be specified.

12.2.1 CONVERT Option

Format:

```
/*$CONVERT*/ or /*$NO CONVERT*/
```

Default: /*\$CONVERT*/

Description:

The convert option may be used to prevent automatic type conversion on the arguments in a function call. Normally, arguments are converted according to C's defined conversion rules for expressions. For example, char is automatically converted to int and float is automatically converted to double. Therefore, it is normally not possible to call a function that has char or float arguments because it is not possible to pass a char or float value. When the convert option is turned off, the compiler does not generate code to perform type conversions on function arguments. Then any type of value may be passed to a function.

Example:

```
printvalue(c, f)
char    c;
float   f;
{
    printf("%c %f", c, f);
}

main()
{
    char c = 'a';
    float f = 1.0;
    /*$NO CONVERT*/
    printvalue(c, f);
    /*$CONVERT*/
}
```

12.2.2 LIST Option

Format:

```
/*$LIST*/ or /*$NO LIST*/
```

Default: /*\$LIST*/

Description:

The list option may be used to turn the compiler listing on and off.

Example:

```
/*$NO LIST*/
#include "stdio"
/*$LIST*/

main()
{
    puts("Do not show listing of stdio");
}
```

12.2.3 LISTMACRO Option

Format:

```
/*$LISTMACRO*/ or /*$NO LISTMACRO*/
```

```
Default: /*$NO LISTMACRO*/
```

Description:

The `listmacro` option may be used to cause the compiler to generate the expansions of macros on the compiler listing. When this option is on, all lines that contain a macro will be followed by the expansion of the line.

Example:

```
/*$LISTMACRO*/  
  
#include "stdio"  
  
main()  
{  
    int c;  
    while ((c=getchar()) != EOF)  
        putchar(c);  
}
```

12.2.4 NESTCMNT Option

Format:

```
/*$NESTCMNT*/ or /*$NO NESTCMNT*/
```

```
Default: /*$NO NESTCMNT*/
```

Description:

The `nestcmnt` option may be used to allow comments to be nested.

Example:

```
/*$NESTCMNT*/  
/* /* This is a comment */ within a comment */
```

12.2.5 PAGESIZE Option

Format:

```
/*$PAGESIZE n*/  
Default: /*$PAGESIZE 60*/
```

Description:

The pagesize option may be used to set the number of program lines that are printed on each page of the compiler listing.

Example:

```
/*$PAGESIZE 10*/  
#include "stdio"  
main()  
{  
    /* There should be 10 program lines per page */  
}
```

12.2.6 SIGNEXT Option

Format:

```
/*$SIGNEXT*/ or /*$NO SIGNEXT*/  
Default: /*$NO SIGNEXT*/
```

Description:

The signext option may be used to cause characters to be sign extended when converted to integers in expressions. This may be useful when using variables of type char for numeric calculations. Sign extension means that if the char value is negative, then the conversion to int will also be negative. Without sign extension, conversion of char to int will always produce a positive result.

Example:

```
main()
{
    char c = -1;
    printf("without sign extend: %d\n", c);
    /*$SIGNEXT*/
    printf("with sign extend: %d", c);
}
```

12.2.7 UPPERCASE Option

Format:

```
/*$UPPERCASE*/ or /*$NO UPPERCASE*/
```

Default: /*\$UPPERCASE*/

Description:

The uppercase option controls the case of function names in the object code. When the option is on, the compiler converts all function names to upper case in the object code. When the option is off, the compiler outputs the function names in the same case as they appear in the program. The main function should be output in upper case letters since the C library references it in upper case. This can be changed by recompiling the C library with the uppercase option turned off.

Example:

```
/*$NO UPPERCASE*/

function()
{
}

/*$UPPERCASE*/

main()
{
}
```

12.2.8 WIDELIST Option

Format:

```
/*$WIDELIST*/ or /*$NO WIDELIST*/
```

Default: /*\$NO WIDELIST*/

Description:

The widelist option may be used to cause the addresses of the generated code to be printed on the compiler listing. All addresses are relative to the beginning of a function. This can be useful as a debugging aid when used in conjunction with the linkload utility. When a program terminates with a runtime error, the address of the last instruction executed is displayed on the screen. By using the symbols command of the linkload utility, the starting address of each function may be found. Separately compiled files must be loaded in the same order as they were when executing the program for this to be effective. By comparing the terminating address with the starting address of each function, the function that was executing when the termination occurred may be determined. Subtracting the starting address of this function from the terminating address gives the address relative to the beginning of the function. This can be compared to the compiler listing to determine the approximate line in the function where the error occurred.

Example:

```
/*$WIDELIST*/
one()
{
    int a = 1;
    return a;
}

main()
{
    printf("One = %d ", one());
}
```

12.2.9 ZERO Option

Format:

```
/*$ZERO*/ or /*$NO ZERO*/
```

Default: /*\$NO ZERO*/

Description:

The zero option sets a runtime flag that effects two things. When the zero option is on, all local variables in a function are initialized to 0 when the function is called. Also, the memory allocated by the standard `calloc` function is initialized to 0. The zero option only has an effect when the main function is compiled.

Example:

```
/*$ZERO*/  
main()  
{  
    char *calloc(), *ptr;  
    int i;  
    ptr = calloc(1, 30);  
    for (i=0; i<30; i++) printf("%d", *ptr++);  
}
```

Appendix A
Error Messages

A.1 Compiler Error Messages

- 1 Identifier expected
- 2 Identifier expected in a type declaration
- 4 ')' expected
- 5 ':' expected
- 6 Illegal symbol
- 7 Invalid preprocessor statement
- 8 Unexpected end of file during declaration definition
- 9 Right braces expected
- 12 Right bracket ']' or '.' expected
- 14 ';' expected
- 15 Integer expected
- 16 '=' expected
- 17 Statement expected
- 18 Closing single quote expected
- 19 Invalid character constant
- 20 ',' expected
- 22 Expression or ';' expected
- 24 Expression expected
- 25 Left parenthesis expected
- 26 WHILE keyword expected
- 27 Lvalue expected (invalid expression on left hand side of assignment)
- 31 Left braces expected or semi-colon missing on function declaration
- 33 Exponential part of floating point number expected
- 36 Lvalue expected for & and * unary operators
- 37 '{' not allowed, contents skipped
- 38 Matching '}' to error 37, text between has been skipped
- 50 Constant expected
- 67 Name of typedef expected
- 68 Structure contains a reference to itself
- 69 Bitfield too wide
- 70 Bitfields must be int or unsigned
- 71 Invalid reference to bitfield
- 72 Too many parameters in macro invocation
- 73 #ENDIF without a matching #IF
- 74 Unexpected eof with unclosed #IF statment
- 75 Preprocessor command expected
- 76 Missing parameter in macro invocation
- 77 Error opening #include file

79 End of file within a comment (missing */)
80 Open comment within a comment
81 Unknown option
101 Identifier already defined
104 Undeclared identifier
114 Null array size allowed for first dimension only
119 Static function definition must precede use
121 Pointer to a function expected
122 ';' not allowed before '{' in function definition
129 Type conflict of operands in an expression
130 Structure assignment with structures of different sizes
131 | or & do not apply to float or double operands
133 Pointer arithmetic requires int
134 Illegal type of operands
137 Incompatible pointers
138 Type of variable is not array
140 Type of variable is not structure or union
141 Type of variable is not pointer
143 Constant expression contains illegal operator
152 No such field in this structure
154 Function parameter list expected
158 Redefinition of a global variable
160 Function definition within a function illegal
162 Function definition has more than one variable in declaration list
164 Parameter declaration expected
165 Label already defined
167 This symbol is not a label
168 Label not defined
170 Too many items in initializer
171 Initializer contains list to initialize single item
172 Type not compatible in initializer
173 Initializer not allowed for this class
174 Initialization of bitfields not allowed on statics & globals
175 Can not be initialized by a string constant
176 & operator does not apply
177 Pointer required for & in initializers
183 Switch selector must be int, unsigned or char
187 Array subscript must be int
188 Invalid type in operands of an op= or = operator
189 Operand to & must be a variable
199 Feature not implemented
202 String constant cannot span lines
203 Integer constant too large
251 Too many errors

A.2 Runtime Error Messages

01) OUT OF STACK

cause: insufficient amount of stack available
cure : If compiling : specify more stack
 CC <stack> file or
 split program into more files
 If executing
 with RUNC : specify more stack space
 RUNC file stack
 with LINKLOAD : specify more stack space
 at the stack prompt

02) OUT OF HEAP

cause: insufficient amount of heap available
cure : If compiling : specify less stack
 CC <stack> file or
 split program into more files
 If executing
 with RUNC : specify less stack space
 RUNC file stack
 with LINKLOAD : specify less stack space
 at the stack prompt

03) BAD POINTER

cause: damaged object file or error in program that
 causes executing code to be overwritten with data
cure : If executing one of the supplied system files,
 restore defective command file from the original
 master disk.
 If executing a user written program,
 check all pointer usage and array indexing.
 Check for a return of NULL when calling the
 calloc function.

04) BAD LEVEL

see error 03

05) DIVIDE BY 0

cause: an integer or real divide operation with a divisor
 of 0
cure : prevent divisor from becoming 0

06) UNDEFINED PCODE

see error 03

07) INVALID SET (Pascal Only)

cause: set operation results in set with more than 256
 members

- 08) BAD RUNTIME CALL
see error 03
- 09) IO ERROR
cause: 1 - file does not exist
2 - disk is full
3 - bad disk or hardware
cure: 1 - specify correct file name
2 - clear some space on the disk
3 - run diagnostics
- 10) RANGE CHECK (Pascal Only)
cause: indexing an array with a subscript out of range
cure : locate and fix indexing into the array
- 11) BAD DIGIT IN NUMBER
cause: attempt to read or DECODE an invalid number
cure : make sure all numbers read or decoded are legal numbers
- 12) PUT ERROR (Pascal Only)
cause: attempt to output an undefined file buffer variable
cure: assign a proper value to the file buffer variable
- 13) OVERFLOW
cause: a real arithmetic calculation overflows
cure : limit real numbers to the maximum size
- 15) UNDERFLOW
cause: a real divide operation causes underflow
cure : limit real numbers to the minimum non-zero size
- 16) LOG NEGATIVE
cause: attempt to take the natural log of a number ≤ 0
cure : log is valid for positive numbers only
- 17) SQRT, X^Y NEGATIVE
cause: attempt to take the square root of a negative number
or attempt to raise a negative number to a real power
cure : square root is valid only for numbers ≥ 0 and
only positive numbers may be raised to a real power
- EB) ATTEMPT TO WRITE TO INPUT FILE
cause: passing an input file pointer to
an output function
- EC) FILE NOT OPEN
cause: attempt to read or write an unopened file
cure : open the file using fopen
- ED) ATTEMPT TO READ OUTPUT FILE
cause: passing an output file pointer to
an input function

EE) NO MEMORY FOR FILE BUFFER
cause: not enough space for file buffer in heap
cure : execute program using less stack

Appendix B

ASCII Table

Decimal	Octal	Hex	Graphic	Name
0	000	00	^@	NUL (used for padding) <null>
1	001	01	^A	SOH (start of header)
2	002	02	^B	STX (start of text)
3	003	03	^C	ETX (end of text)
4	004	04	^D	EOT (end of transmission)
5	005	05	^E	ENQ (enquiry)
6	006	06	^F	ACK (acknowledge)
7	007	07	^G	BEL (bell or alarm)
8	010	08	^H	BS (backspace) <bs>
9	011	09	^I	HT (horizontal tab) <tab>
10	012	0A	^J	LF (line feed) <lf>
11	013	0B	^K	VT (vertical tab)
12	014	0C	^L	FF (form feed, new page) <ff>
13	015	0D	^M	CR (carriage return) <cr>
14	016	0E	^N	SO (shift out)
15	017	0F	^O	SI (shift in)
16	020	10	^P	DLE (data link escape)
17	021	11	^Q	DC1 (device control 1, XON)
18	022	12	^R	DC2 (device control 2)
19	023	13	^S	DC3 (device control 3, XOFF)
20	024	14	^T	DC4 (device control 4)
21	025	15	^U	NAK (negative acknowledge)
22	026	16	^V	SYN (synchronous idle)
23	027	17	^W	ETB (end transmission block)
24	030	18	^X	CAN (cancel)
25	031	19	^Y	EM (end of medium)
26	032	1A	^Z	SUB (substitute)
27	033	1B	^[ESCAPE (alter mode, SEL) <esc>
28	034	1C	^\	FS (file separator)
29	035	1D	^]	GS (group separator)
30	036	1E	^^	RS (record separator)
31	037	1F	^_	US (unit separator)
32	040	20	' '	space or blank <sp>
33	041	21	!	exclamation mark
34	042	22	"	double quote
35	043	23	#	number sign (hash mark)
36	044	24	\$	dollar sign
37	045	25	%	percent sign
38	046	26	&	ampersand sign
39	047	27	'	single quote (apostrophe)

40	050	28	(left parenthesis
41	051	29)	right parenthesis
42	052	2A	*	asterisk (star)
43	053	2B	+	plus sign
44	054	2C	,	comma
45	055	2D	-	minus sign (dash)
46	056	2E	.	period (decimal point)
47	057	2F	/	(right) slash
48	060	30	0	numeral zero
49	061	31	1	numeral one
50	062	32	2	numeral two
51	063	33	3	numeral three
52	064	34	4	numeral four
53	065	35	5	numeral five
54	066	36	6	numeral six
55	067	37	7	numeral seven
56	070	38	8	numeral eight
57	071	39	9	numeral nine
58	072	3A	:	colon
59	073	3B	;	semi-colon
60	074	3C	<	less-than sign
61	075	3D	=	equal sign
62	076	3E	>	greater-than sign
63	077	3F	?	question mark
64	100	40	@	at sign
65	101	41	A	upper-case letter ABLE
66	102	42	B	upper-case letter BAKER
67	103	43	C	upper-case letter CHARLIE
68	104	44	D	upper-case letter DELTA
69	105	45	E	upper-case letter ECHO
70	106	46	F	upper-case letter FOXTROT
71	107	47	G	upper-case letter GOLF
72	110	48	H	upper-case letter HOTEL
73	111	49	I	upper-case letter INDIA
74	112	4A	J	upper-case letter JERICH0
75	113	4B	K	upper-case letter KAPPA
76	114	4C	L	upper-case letter LIMA
77	115	4D	M	upper-case letter MIKE
78	116	4E	N	upper-case letter NOVEMBER
79	117	4F	O	upper-case letter OSCAR
80	120	50	P	upper-case letter PAPPA
81	121	51	Q	upper-case letter QUEBEC
82	122	52	R	upper-case letter ROMEO
83	123	53	S	upper-case letter SIERRA
84	124	54	T	upper-case letter TANGO
85	125	55	U	upper-case letter UNICORN
86	126	56	V	upper-case letter VICTOR
87	127	57	W	upper-case letter WHISKY
88	130	58	X	upper-case letter XRAY
89	131	59	Y	upper-case letter YANKEE
90	132	5A	Z	upper-case letter ZEBRA
91	133	5B	[left square bracket
92	134	5C	\	left slash (backslash)
93	135	5D]	right square bracket
94	136	5E	^	uparrow (carat)

95	137	5F		underscore
96	140	60	`	(single) back quote
97	141	61	a	lower-case letter able
98	142	62	b	lower-case letter baker
99	143	63	c	lower-case letter charlie
100	144	64	d	lower-case letter delta
101	145	65	e	lower-case letter echo
102	146	66	f	lower-case letter foxtrot
103	147	67	g	lower-case letter golf
104	150	68	h	lower-case letter hotel
105	151	69	i	lower-case letter india
106	152	6A	j	lower-case letter jericho
107	153	6B	k	lower-case letter kappa
108	154	6C	l	lower-case letter lima
109	155	6D	m	lower-case letter mike
110	156	6E	n	lower-case letter november
111	157	6F	o	lower-case letter oscar
112	160	70	p	lower-case letter pappa
113	161	71	q	lower-case letter quebec
114	162	72	r	lower-case letter romeo
115	163	73	s	lower-case letter sierra
116	164	74	t	lower-case letter tango
117	165	75	u	lower-case letter unicorn
118	166	76	v	lower-case letter victor
119	167	77	w	lower-case letter whisky
120	170	78	x	lower-case letter xray
121	171	79	y	lower-case letter yankee
122	172	7A	z	lower-case letter zebra
123	173	7B	{	left curly brace
124	174	7C		vertical bar
125	175	7D	}	right curly brace
126	176	7E	~	tilde
127	177	7F	<rubout>	DEL

Appendix C

Differences from Kernighan and Ritchie

Extensions

- (1) Structures may be assigned.
- (2) Locally declared arrays and structures may be initialized.
- (3) Nested comments are allowed via a compiler option.
- (4) Char and float values may be passed to a function via a compiler option that prevents automatic type conversion of the arguments in a function call.
- (5) Local variables may be initialized to 0 via a compiler option.

Restrictions

- (1) The address of (&) an array element may not be used as an initializer.
- (2) External float and double variables may not be initialized.

Other Differences

- (1) Reserved words may be lower case, upper case, or mixed case. K & R says that reserved words must be lower case.

Index

#define 150
#else 154
#endif 151, 152, 153
#if 153
#ifdef 151
#ifndef 152
#include 149
#line 155
#undef 151
_exit 148
_main function 97
abs 142
addition operator 35
address of, & 53
alternate symbols 7
and operator, logical 40
and, & 48
arguments 13
arithmetic operators 34
array initialization 75
array of characters 73
arrays 72
assignment operator 34
assignment operators 51
associativity 31
atan 142
atof 136
atoi 135
auto variables 24
bit fields 83
bitwise operators 46
Braces 9
break statement 92
calloc 139
cast operator 55
cfree 141
Character Constants 3
character functions 123
character I/O 100
character variables 19
closing files 97
comma operator 56
Comments 8
compiler options 156
conditional expression 58
conditional statements 87
Constants 2
contents of, * 53

continue statement 93
conversion functions 135
CONVERT option 156
cos 143
decrement, -- 43, 44
division operator 36
do-while statement 91
double precision 23
dstos 134
dynamic memory 139
dynamic strings 133
else statement 88
equality operator 39
equality operators 37
Escape Sequence 3
exclusive or, ^ 49
exit 147
exp 144
expression, ?: 58
extern variables 25
fclose function 99
fgets function 105
float type variables 22
Floating Point Constants 3
fopen function 98
for statement 92
formatted I/O 107
formatted input 108
formatted output 111
fprintf 118
fputs function 106
fscanf 117
ftoa 137
function arguments 63
function body 63
function call 67
function declarations 64
function definition 61
function names 62
function pointers 67
function types 62
functions 11, 61
getc function 101
getchar function 100
gets function 104
global variable declarations 64
global variables 16
goto statement 94
greater than operator 39
greater than or equal to operator 39
Identifiers 1
if statement 87
increment, ++ 43, 44
inequality operator 39
integer constants 2
integer variables 20

isalpha 123
isdigit 124
islower 125
isspace 125
isupper 126
itoa 137
labels 94
less than operator 39
less than or equal to operator 39
LIST option 157
LISTMACRO option 158
local variable declarations 63
local variables 15, 24
log 144
logical operators 40
long integers 21
looping statements 90
main function 11, 65
modulo operator 37
multiplication operator 36
negate, ~ 46
NESTCMNT option 158
nested blocks 65
not equal operator 39
not operator, logical 41
null statement 95
opening files 97
operator grouping 31
operator precedence 31
operator table 33
Operators 7
or operator, logical 41
or operator, | 50
PAGESIZE option 159
pointers 71
printf 116
putc function 102
putchar function 101
puts function 105
recursion 68
register variables 28
relational operators 37
Reserved Words 6
return statement 95
scanf 115
Semicolon 8
shift left, << 48
shift right operator 47
short integers 20
SIGNEXT option 159
sin 145
sizeof operator 54
sprintf 121
sqr 145
sqrt 146
sscanf 119

statement labels 94
static functions 66
static variables 27
stderr 97
stdin 97
stdout 97
stods 133
storage classes 24
strcat 131
strcmp 130
strcpy 129
string 73
String Constants 5
string functions 128
string I/O 104
strlen 129
strsave 132
structure arrays 82
structure bit fields 83
structure initialization 80
structure member operator 56, 80
structure pointer operator 57, 81
structure pointers 81
structure tags 77
structures 77
subtraction operator 35
switch 89
Symbolic Constants 4
termination 147
Terminology 10
tolower 127
toupper 128
transcendental functions 141
type conversions 42
typedef 29
unary minus operator 36
ungetc function 103
unions 77, 84
unsigned integers 22
UPPERCASE option 160
void function type 62
while statement 90
WIDELIST option 161
ZERO option 162

Table of Contents

Chapter 1 Using the Optimizer	3
1.1 When to Use the Optimizer	3
1.2 How to Use the Optimizer	3
1.2.1 Short Form	4
1.2.2 Long Form	4
1.3 How the Optimizer Works	5
Chapter 2 Using the Code Generator	7
2.1 When to Use the Code Generator	7
2.2 How to Use the Code Generator	8
2.2.1 Short Form	8
2.2.2 Long Form	9
2.3 How the Code Generator Works	9
Chapter 3 Mixed Mode Operation	11
3.1 When to Use Mixed Mode	11
3.2 How to Use Mixed Mode	12
Chapter 4 Object Format	13
4.1 Address Independent	13
4.2 ASCII Format	13
4.3 Object Code Tags	14
4.4 Splitting Object Modules	15
Chapter 5 Assembly Language	17
5.1 Assembly Language Structure	17
5.2 Assembly Language Format	18

Introduction

The advanced development package (ADP) is a software tool which adds a great deal of power and versatility to the language system. The advanced development package consists of two programs. One program is an optimizer which reduces the size of programs. The other is a code generator which increases the speed of programs. The combination gives the programmer the ability to customize each application program, allowing for maximum utilization of the systems capabilities.

The need for the optimizer occurs when writing large programs. All programs require memory to store instructions and memory to store data. Large programs require a lot of memory to store instructions. The memory used for storing instructions subtracts from the memory available for storing data (ie. the more memory used for storing instructions, the less available for storing data). The optimizer's purpose is to reduce the amount of memory used by the instructions in order to make more memory available for storing data.

The need for the code generator occurs when execution speed is important. The compiler translates source programs to instructions known as pseudo code (p-code for short). The computer cannot directly execute instructions in p-code form. Instead they are executed by another program known as an interpreter. Maximum execution speed can be achieved by translating programs to machine code (the form which the computer hardware can understand and execute directly without interpretation). The purpose of the code generator is to translate p-code instructions to machine instructions. This provides a method for achieving maximum execution speed.

The addition of the advanced development package provides the programmer with a very flexible language system which offers a unique ability. This is the ability to mix p-code with machine code. P-code has the advantage of compactness while machine code has the advantage of speed. The ability to mix the two makes it possible to customize application programs in order to achieve optimum performance. The bottle neck areas of a program may be translated to machine code for maximum speed while the rest of the program can be left in p-code form. This allows programs to benefit from both compactness and speed.

Chapter 1

Using the Optimizer

The optimizer is a program which takes the compiler generated p-code as input and outputs an optimized form of the same p-code. The purpose of the optimization is to make the p-code more compact. The difference in size of the optimized versus non-optimized p-code is dependent on the types of language features utilized by the original source program. Typically, the percent reduction in size due to optimization will fall in the range of 10 to 30 percent. This size reduction is sometimes very important. By making the program smaller, there is more room for data. Often times, it will enable the execution of a program that otherwise would run out of memory.

1.1 When to Use the Optimizer

The optimizer should be used any time program size is an important factor. A programs memory requirements are determined by the number of executable instructions and by the number and sizes of the variables used. The factor that the optimizer addresses is the number and length of instructions. The greatest benefit will then be realized when optimizing long programs (>200 lines).

1.2 How to Use the Optimizer

Any p-code object file may be used as input to the optimize utility. The compiler generates an OBJ extension as the default for p-code object files. Whole programs or separately compiled parts of a program may be optimized. In either case, simply compile the source program and then run the compiler generated p-code through the optimizer.

NOTE: Only p-code object files may be optimized. Do not attempt to optimize command files or files generated by the code generator.

The optimize utility is stored as a command file and therefore may be executed simply by typing OPTIMIZE from the top level of the operating system. It has two forms for input, a short form and a long form.

1.2.1 Short Form

```
OPTIMIZE file-name
```

The file-name should not include an extension. The optimizer appends the default extension OBJ to the file name. The output of the optimizer (the optimized pcode) is placed in a file of the same name but with the extension OPT. The OPT file can be used any place that an OBJ file is used.

NOTE: The file-name may include a drive specification.
When a drive is specified, the OPT file is placed on the same drive as the OBJ file.

1.2.2 Long Form

```
OPTIMIZE  
LISTING = list-file  
INP_OBJ = obj-file  
OUT_OPT = opt-file
```

The long form requires that you enter the full file name, including extension, for both the input object file (obj-file) and output object file (opt-file). The LISTING will show the name of each separate module in the input p-code object file as it is processed. After each name will appear its original size in bytes followed by its optimized size in bytes. The LISTING may be directed to a file or device. Simply typing the <ENTER> key will direct the listing to the screen.

At completion, the optimize utility will display on the listing the size of the non-optimized p-code used as input and the optimized p-code generated as output.

```
ORIGINAL LENGTH = size in bytes  
OPTIMIZED LENGTH= size in bytes
```

1.3 How the Optimizer Works

The optimizer is a program which contains a loader for loading p-code object files. The loader loads and operates on one module (procedure and/or function) at a time, maintaining context as it operates on each individual module. The p-code instructions are analyzed to determine whether or not they may be compressed into shorter instructions.

Since the compiler is one pass, it must generate some branch and addressing instructions without knowing the actual displacements. This makes it necessary to allocate two byte operands for unknown displacements in order to handle all cases. However, in many cases the displacements can be specified using only one byte. The optimizer looks for such cases and compresses the p-code instructions in order to take advantage of the need for only a single byte operand.

The optimizer also looks for other types of situations where compression of instructions is possible. For example, all multiply by two instructions are converted to add instructions. In certain cases, consecutive instructions can cancel one another out (eg. an increment followed by a decrement). The optimizer eliminates such cases. The optimizer also performs constant folding (ie. it replaces arithmetic operations involving only constant operands with a single constant value). For example, $2+2$ would be replaced by the single constant 4.

Chapter 2

Using the Code Generator

The code generator is a program which translates p-code instructions to machine instructions. Any compiler generated p-code object file or optimized p-code file may be used as input to the code generator. Whole programs or separately compiled parts of programs may be translated (codegened) to machine instructions to increase execution speed. The speed increase realized from code generation is dependent on the nature of the program. Typically, codegened programs will gain a factor of 3 to 5 times increase in speed over that of pure p-code programs.

2.1 When to Use the Code Generator

The code generator increases execution speed by translating p-code instructions to machine instructions. Since each p-code instruction is equivalent to several machine instructions, code generation also causes an increase in size. Therefore, the decision of whether or not to perform code generation on a program must not only be based on speed requirements, but also on program size. Typically, code generation will cause the size of the object code to increase by a factor of 2 to 3 over that of pure p-code.

The execution speed of most programs will be adequate even when left in p-code form. However, programs which perform lots of calculations within loops will benefit significantly from code generation. Also, when a program contains one or more routines that are frequently called, code generation on these sections of the program can provide significant improvement in execution speed. For example, the scanner of the compiler is a routine which reads the text of a source program and distributes it to other parts of the compiler. Since it is called frequently, much of the time spent during a compile is inside this one routine. Code generation of the scanner can increase compile speed significantly. By selecting the parts of a program which most effect speed and performing code generation only on those parts, speed can be increased without significant increase in size.

The determination of whether or not to codegen a program can be made by observation. First run the program in p-code form. If execution speed is observed to be slow, the next step is to determine whether or not to codegen the whole program or selected parts of the program. As a general rule, small programs should be totally codegened. The size increase for small programs will probably be insignificant. However, for large programs, the size increase may be very significant.

For large programs, a factor of 2 to 3 increase in object code size will significantly reduce the amount of memory left for the program data area (stack and heap). In cases where the size increase would not allow enough room for data area, selected routines should be compiled separately. The routines selected should be the ones which most effect execution speed. These routines may then be codegened and linked to the main program. This process will allow for an increase in speed without causing the size to increase to a level that prevents the program from being executed.

2.2 How to Use the Code Generator

Any compiler generated or optimized p-code object file can be used as input to the code generator. The compiler generates files with the default extension of OBJ. The optimizer generates files with a default extension of OPT. Whole programs or separately compiled routines may be codegened. In either case, simply compile the source, optionally optimize the compiler output, and then run the p-code object file through the code generator utility.

The code generator utility is stored as a command file and is therefore executed simply by typing CODEGEN from the top level of the operating system. It also has two forms.

2.2.1 Short Form

```
CODEGEN file-name
```

The file-name should not include an extension. The code generator appends the default extension OBJ to the file name. The output of the code generator is placed in a file of the same name but with the extension COD. The COD file may then be used with the other system utilities. However, do not attempt to optimize a COD file. The COD files contain machine instructions and the optimizer accepts only p-code instructions.

NOTE: File-name may also include a drive specifier.

When a drive is specified, the COD file is placed on the same drive as the OBJ file. The file named CODEINIT must be on line when CODEGEN is executed.

2.2.2 Long Form

```
CODEGEN
INP_OBJ = obj-file or opt-file
OUT_COD = cod-file
DO YOU WANT ASSEMBLY LANGUAGE SOURCE? (Y,N): y or n
```

The long form requires that you enter the full file name, including extension, for both input and output files. If assembly language output is desired, answer Y to the last prompt, otherwise answer N. If assembly language output is requested, the following prompt will appear.

```
SOURCE =
```

The additional assembly language output will be directed to the file specified. The assembly language output is discussed in chapter 5.

NOTE: File names may include drive specifiers.

2.3 How the Code Generator Works

The code generator is a program which contains a loader for loading p-code object files. The code generator loads one module (procedure or function) at a time and translates the p-code instructions to machine instructions. As noted earlier, a p-code instruction is equivalent to several machine instructions, so the translation process will increase the total number of instructions.

There are a few p-code instructions which perform very complex functions. To perform equivalent functions in machine code would require a very large number of instructions. Therefore, a few selected p-code instructions are not translated to machine instructions. They are left in p-code form and executed as subroutine calls to assembly language routines within the interpreter. Handling complex functions in this manner prevents the COD file from becoming as large as it would with complete translation.

Chapter 3

Mixed Mode Operation

Through the use of the linking loader (the LINKLOAD utility), pure p-code object (OBJ or OPT) files may be linked with codegened (COD) files. Executable programs (command files) may then be built which contain mixed instructions, both p-code and machine code. This ability is important when writing large programs. It allows you to select and codegen only those parts of a program which most effect the speed of execution. The remaining parts of the program can be left in pcode form. This mixed mode operation allows you to increase execution speed without significantly increasing program size.

3.1 When to Use Mixed Mode

The use of mixed mode is usually not important until you start developing large programs. Small programs can be totally codegened without the size increase becoming a significant factor. However, completely codegening large programs (>500 lines) may cause a size increase which will prevent the program from being executed. The code size of the program can become so large that there is no longer enough room for data storage. This of course depends on the data storage requirements of the program.

When developing large programs, you should not consider code generation until after executing the program in p-code form. Observe the execution speed to determine whether or not it is adequate for your application. If not, the next step is to decide what areas of the program are most effecting the speed. Long loops are typical areas of a program where most of the execution time is spent. Another area might be a low level routine or several routines that are called frequently throughout a program.

After deciding which areas of the program are effecting execution speed the most, separate them from the rest of the program and codegen them. The selection and separation process is easiest if the program is well modularized. That is, the program is already segmented into modules, each performing a distinct and well defined function.

3.2 How to Use Mixed Mode

Once the speed critical routines of a program have been selected for codegening, they must be separated from the rest of the program. The separated routines should then be compiled separately from the remainder of the program. Once the separated routines are compiled, they may be codegened and then linked to the remainder of the program using the linking loader.

The process for mixed mode operation is summarized by the following list of steps.

- 1) Select the areas which most effect program speed.
- 2) Separate the selected parts of the program from the remainder of the program. Any selected routines should be placed in one or more separate files.
- 3) Compile all parts of the program.
- 4) CODEGEN the parts of the program which were selected to increase execution speed.
- 5) LINKLOAD all compiled parts of the program together and either run the program or build an executable command file.

Chapter 4

Object Format

All Alcor Systems compilers generate object code in the same format. The object code generated by one compiler can be linked with the object code generated by another. The object code format was designed in this way so that programs written in one language could call routines written in another.

4.1 Address Independent

The pseudo-code (p-code) generated by the compilers is address independent. That is, it contains no absolute memory addresses and can execute without change when loaded anywhere in memory. All branch and call instructions are performed relative to the program counter. Since functions and/or procedures are compiled into separate modules, calculation of these relative addresses must be done when the object code is loaded. The object format supports external references that are program counter relative.

4.2 ASCII Format

The object code is stored in standard ASCII format. This causes an object file to require more space than one stored in binary format. However, the advantage of being able to edit object files or transmit them across a modem is greater than the disadvantage of creating slightly larger object files.

4.3 Object Code Tags

The object code is tagged hexadecimal and is emitted in a line oriented stream. Each item in the object file begins with a tag which is usually an upper case letter. The tag defines the type of item and the number and size of the fields to follow. Tags are followed by one or more fields that specify the information to be loaded. Three types of fields exist. Bytes are specified with a two character hexadecimal number. Words consist of a four character hexadecimal number with the most significant byte first. Labels consist of eight character names that are the names of external symbols.

Following is a table which lists all the tags used in an object file. All tags are followed by one to three fields of information, each field being either a byte, word, or label. The meaning of each tag is also shown.

Tag	Field1	Field2	Field3	Meaning
A	byte			Absolute(non-relocatable byte)
E				End of module
F				End of line
G	word	label		Definition of external symbol
I	word	label		External reference declaration
J	label			Module name
Q	word			Reference to external symbol
M	word	word	label	Definition of common block
N	word			Reference to common
O	word	word		Overlay definition
P	word			Code (PC) origin
K	word			Relative reference to external
W	word			Relocatable word
X	word			Absolute word
Y	word			Entry point definition
:				End of file

4.4 Splitting Object Modules

The compiler generates independent object modules for each function and/or procedure in a program. The following is an example of what an object module looks like.

```
JLOOP      P0000G0000LOOP      A01X0000A38A02A03X0001A15A02A10F
A02A03X2710A07A15A06A2BA4EX0000A03X0001A03X0002A22A03X0003F
A22A03X0004A22A03X0005A22A03X0006A22A03X0007A22A03X0008A22F
A03X0009A22A03X000AA22A03X000BA22A03X000CA22A03X000DA22A03F
X000EA22A03X000FA22A15A04A10A02A30X0002A10A06A27A21AB9P0014F
X0047P005DA3AP0001X0006E
:
```

Each module in an object file begins with the module name. Therefore, it is possible to split a file containing several modules into several files, each containing one module. This is one way to allow code generation of selected modules.

There are two ways to split the object modules. One is to text edit them. The other method is to write a program to split them. A simple program may be written to read an object file. Each time a module is encountered, open a file of the same name as the module and write the module to that file. The file name may have to be slightly changed to remove any illegal file name characters. Each module always begins with a J tag and ends with an E tag.

Once all the modules are separated into different files, selected modules may be input to the code generator and translated to machine instructions. The linking loader may then be used to link the individual modules and build an executable command file.

Chapter 5

Assembly Language

The code generator has the capability of producing assembly language source in addition to object code. It is not necessary in normal circumstances to generate the source, since the object code emitted by CODEGEN is exactly equivalent to the result of assembling the source. The assembly language is provided as a means for the programmer to examine the code produced by the native code generator. In some cases, the programmer may wish to optimize this code by hand and assemble it. It is expected that the need to do this will be rare, since the effort is substantial and the improvements that can be made are minor. If you wish to assemble the source output of CODEGEN, then the Alcor Systems XASM assembler is required.

The source output of CODEGEN is useful to the assembly language programmer who wishes to link assembly language modules to programs and to call them as normal program functions. A technique to accomplish this is to write a dummy function with the same name and calling sequence as the assembly language routine. The actual code can be left out and perhaps replaced by a template that merely accesses the arguments that will be used in assembly language. The dummy function can be compiled and run through the code generator with the source option enabled. Codegen will generate the proper function linkage and will calculate the addresses of variables and arguments referenced in the body of the function. The generated code can then be used as a skeleton for the assembly language that actually implements the functions required. The XASM assembler is required.

5.1 Assembly Language Structure

The assembly language source emitted by CODEGEN is designed for assembly by the Alcor Systems multiprocessor assembler (XASM). This assembler provides the ability to mix Z80 code (or 6502 code, or 1802 code, or 8080 code) with p-code. Essential assembler features include the ability to switch among target processors (Z80 to p-code), the ability to define and reference external symbols (externals are resolved at load time), and the ability to generate p-code addressing modes (program counter relative, stack displacement, etc.).

Each routine in a program is compiled into a separate object module. All symbols, labels, and instructions are local to the module and reference other modules only via explicit external references. Modules begin with a module identification. The module name is the name of the routine truncated to 8

characters. Each routine also contains an external definition of its name. This is signaled with the "DEF" assembler directive (G tag). The DEF statement causes the name and its value to be defined externally so that other modules can call it.

Switching between modes (machine vs. p-code) takes place within the routine. Some features of a compiled language are sufficiently complex that they are implemented with subroutines. Inclusion of the actual code in-line would make the generated code unreasonably large. When complex operations (such as input or output) are performed, the code generator produces a call to a runtime procedure. These runtime procedures are already part of the p-code interpreter. Rather than reference them again (and require another copy), the processor is switched back to p-code mode and the interpreter is allowed to perform the operation.

When in mixed mode, all calling is performed using the p-code interpreter. Since code for each module is separate, and since modules may be split before being loaded, it is unknown whether the module being called is p-code or machine code. Therefore, every module is entered in p-code mode. If the module is machine code, the processor is switched to machine code mode immediately after entry to the module.

5.2 Assembly Language Format

The source code emitted by CODEGEN uses extended 8080 mnemonics. This is done primarily for historical reasons and since the 8080 instruction set more clearly distinguishes instructions by format. Use of 8080 extended mnemonics affects only the source output of codegen, as the Z80 instruction set is used and converted directly to object code by CODEGEN. Each instruction occupies one line. Labels are left justified and begin with a letter. Each instruction has an opcode which is either an 8080 instruction or a Z80 instruction. There are also pseudo-operators (pseudo-ops) that provide instructions to the assembler rather than generating code.

Operands use standard register names. In many cases, the names of the Z80 index registers are merged with the opcode (e.g. PUSHIX pushes the IX index register). This simplifies interpretation by the assembler. Operands may also use symbolic labels and constants. Constants are normally expressed in hexadecimal (base 16) with a leading greater than sign (">") to specify hexadecimal to the assembler.

The following table lists the pseudo operators that are directives to the assembler.

IDT	identifies the module and gives it a name
EQU	defines the value of the label to the result of evaluating the operand
DEF	defines the operand as an external symbol
REF	specifies that the operand is an external symbol that is defined in another module
CSEG	Specifies the name and size of a common block
QLIST	Selects the compact format for the assembler listing
END	Signals the end of the module
ENTRY	Defines an entry point into the module
SETCPU	Selects the processor whose assembly language is being assembled

RADIO SHACK  **A DIVISION OF TANDY CORPORATION**

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA

**91 KURRAJONG ROAD
MOUNT DRUITT, N.S.W. 2770**

BELGIUM

**PARC INDUSTRIEL DE NANINNE
5140 NANINNE**

U. K.

**BILSTON ROAD WEDNESBURY
WEST MIDLANDS WS10 7JN**