

Introduction to

C Programming

S.C. Huí

Prentice
Hall

INTRODUCTION TO C PROGRAMMING

Second Edition

S.C. Hui

Nanyang Technological University



PRENTICE HALL SprintPrint Publishing

Contents

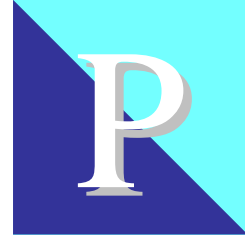
PREFACE	viii
1 COMPUTER SYSTEMS AND C PROGRAMMING	1
1.1 Computer Systems	1
1.2 Computer Software	3
1.3 Programming Languages	5
1.4 Why C Programming?	6
1.5 ANSI C Standard	7
1.6 Structure of a C Program	7
1.7 A Sample C Program	11
1.8 Development of a C Program	13
1.9 Exercises	17
2 PROGRAM DEVELOPMENT	18
2.1 Program Development Process	18
2.2 Program Design	20
2.3 Flowcharts and Pseudocode	21
2.4 Control Structures	22
2.5 Top-down Stepwise Refinement	27
2.6 Implementation	33
2.7 Program Testing	37
2.8 Documentation	38
2.9 Exercises	39
3 DATA TYPES, CONSTANTS AND VARIABLES	41
3.1 Basic Data Types	41

3.2	The Integer Type	42
3.3	The Floating Point Type	43
3.4	The Character Type	45
3.5	Constants	45
3.6	Variables and Declarations	49
3.7	Exercises	51
4	SIMPLE INPUT/OUTPUT	53
4.1	Formatted Input/Output	53
4.2	The printf() Function	54
4.3	The scanf() Function	61
4.4	Character Input/Output	66
4.5	Exercises	69
5	OPERATORS, EXPRESSIONS AND ASSIGNMENTS	70
5.1	Operators	71
5.2	Fundamental Arithmetic Operators	71
5.3	Increment/decrement Operators	72
5.4	Precedence of Operators	74
5.5	Expressions	74
5.6	Assignment Statements	75
5.7	Data Type Conversion	80
5.8	Mathematical Functions	83
5.9	Bit Manipulation	84
5.10	Exercises	88
6	BRANCHING	90
6.1	Relational and Logical Operators	90
6.2	The if Statement	93
6.3	The if-else Statement	95
6.4	The if-else if-else Statement	96
6.5	Nested-if	99
6.6	The switch Statement	102
6.7	The Conditional Operator	106
6.8	Exercises	108

7	LOOPING	110
7.1	Types of Loops	110
7.2	The while Statement	111
7.3	The for Statement	118
7.4	The do-while Statement	124
7.5	The break Statement	127
7.6	The continue Statement	129
7.7	Nested Loops	131
7.8	Exercises	134
8	FUNCTIONS	136
8.1	Function Definition	136
8.2	Function Prototypes	142
8.3	Calling a Function	143
8.4	Call by Value	146
8.5	Pointers	149
8.6	Call by Reference	156
8.7	Recursive Functions	159
8.8	Functional Decomposition	166
8.9	Placing Functions in Different Files	167
8.10	Exercises	169
9	ARRAYS	172
9.1	Array Declaration	173
9.2	Initialization of Arrays	174
9.3	Operations on Arrays	175
9.4	Pointers and Arrays	178
9.5	Arrays as Function Arguments	182
9.6	Sorting Arrays	184
9.7	Searching Arrays	188
9.8	Multidimensional Arrays	192
9.9	Multidimensional Arrays and Pointers	197
9.10	Multidimensional Arrays as Function Arguments	198
9.11	Exercises	207
10	CHARACTER STRINGS AND STRING FUNCTIONS	210
10.1	String Constants	210
10.2	String Variables	211

10.3	String Input	218
10.4	String Output	222
10.5	String Functions	223
10.6	The ctype.h Character Functions	232
10.7	String to Number Conversions	234
10.8	Formatted String Input and Output	235
10.9	Arrays of Character Strings	236
10.10	Command Line Arguments	238
10.11	Exercises	240
11	STRUCTURES, UNIONS AND ENUMERATED TYPES	244
11.1	Structures	244
11.2	Arrays of Structures	249
11.3	Nested Structures	251
11.4	Pointers to Structures	253
11.5	Functions and Structures	255
11.6	The typedef Construct	261
11.7	Unions	262
11.8	Enumerated Data Type	266
11.9	Exercises	268
12	FILE INPUT/OUTPUT	270
12.1	Streams and Buffers	270
12.2	Files	275
12.3	Accessing a File	276
12.4	Character Input/Output	281
12.5	String Input/Output	283
12.6	Formatted File Input/Output	285
12.7	Binary and Block I/O	286
12.8	Random Access	291
12.9	Other I/O Functions	296
12.10	Exercises	297
13	STORAGE CLASSES	299
13.1	Storage Classes	299
13.2	Storage Class auto	302
13.3	Storage Class extern	303
13.4	Storage Class static	304

13.5	Storage Class register	305
13.6	Local and Global Variables	306
13.7	Nested Blocks with the Same Variable Name	308
13.8	Storage Classes for Multiple Source Files	309
13.9	Storage Classes for Functions	311
13.10	ANSI C Type Qualifiers	312
13.11	Exercises	314
14	THE C PREPROCESSOR	316
14.1	The C Preprocessor	316
14.2	The #define Directive	319
14.3	Macros with Arguments	322
14.4	File Inclusion	327
14.5	Conditional Compilation	329
14.6	Predefined Preprocessor Macros	334
14.7	Other Directives and Operators	335
14.8	Exercises	336
APPENDIX A	ASCII CHARACTER SET	338
APPENDIX B	ANSWERS TO EXERCISES	339
INDEX		361



Preface

Introduction

This book introduces the basic concepts of C programming. C has a number of advantages over other programming languages. This includes flexibility, portability, efficiency and modularity. It is also very easy to learn. Therefore, C is commonly introduced as the programming language for first year engineering students. The objectives of this book are the following:

- To teach students the concepts of C programming language.
- To present the program development process, and demonstrate the application of techniques for analyzing elementary algorithmic problems and implementing the program design in C language.
- To illustrate the use of good programming practice to the development of programs.

Organization of the Book

Chapter 1 presents an introduction to computer systems and programming languages. In particular, the fundamental concepts on C programming are given. The program development process is discussed in Chapter 2. Chapter 3 describes the concepts on data types, constants and variables. Chapter 4 discusses simple input/output functions such as **scanf()** and **printf()** functions. Chapter 5 covers the concepts on operators, expressions and assignments. Chapter 6 discusses the use of branching statements such as **if**, **if-else** and **if-else if-else** statements. Chapter 7 shows the use of looping statements such as **for**, **while** and **do-while** loops. Chapter 8 discusses the use of functions for designing larger programs. Chapter 9 contains a discussion on arrays. Chapter 10 deals with character strings and string functions. Structures and other data types

such as **enum** and **typedef** are discussed in Chapter 11. Chapter 12 covers the topic on file input/output. Chapter 13 discusses the storage classes of C. The C preprocessor and macros are presented in Chapter 14.

Acknowledgement

I would like to acknowledge the help from Dr. Huang Shell Ying in providing some of the programming examples. I would also like to thank Dr. Edmund Lai and Dr. Alvis Fong for reviewing the book and providing useful suggestions. I am grateful for my postgraduate student, Lee Pui Yen, for drawing the diagrams and verifying the solutions for the programming examples in the book. Furthermore, I am indebted to the School of Computer Engineering, Nanyang Technological University, for providing me time and support for writing this book. Finally, I thank my wife, Xiao Bing, and children, Kenny and Vincent, for their patience and support while this book was in preparation.

Siu Cheung Hui
asschui@ntu.edu.sg
Nanyang Technological University
School of Computer Engineering
Singapore



Computer Systems and C Programming

In this chapter, we introduce computer systems and describe the major components of a computer system. We focus our discussion on computer software and programming languages, particularly the high-level language C, and the various advantages of using C as a programming language. The basic structure of a C program and the various components for a C program are discussed. A sample C program is then used to illustrate these basic components. The program development cycle for creating, compiling and executing a C program is described.

This chapter covers the following topics:

- 1.1 Computer Systems
 - 1.2 Computer Software
 - 1.3 Programming Languages
 - 1.4 Why C Programming?
 - 1.5 ANSI C Standard
 - 1.6 Structure of a C Program
 - 1.7 A Sample C Program
 - 1.8 Development of a C Program
-

1.1 Computer Systems

A computer system mainly consists of hardware and software. Hardware includes the physical devices that form the computer system. Software is a computer

program that is executed on top of hardware to perform a specific task in order to solve a problem.

The architecture of modern computer systems varies greatly from type to type. Basically, there are four types of computer systems: mainframes and minicomputers, workstations, personal computers and portable computing devices. For most computer systems, they have the following components, namely Central Processing Unit (CPU), Main Memory, Secondary Storage, and Input and Output (I/O) Devices, as shown in Figure 1.1.

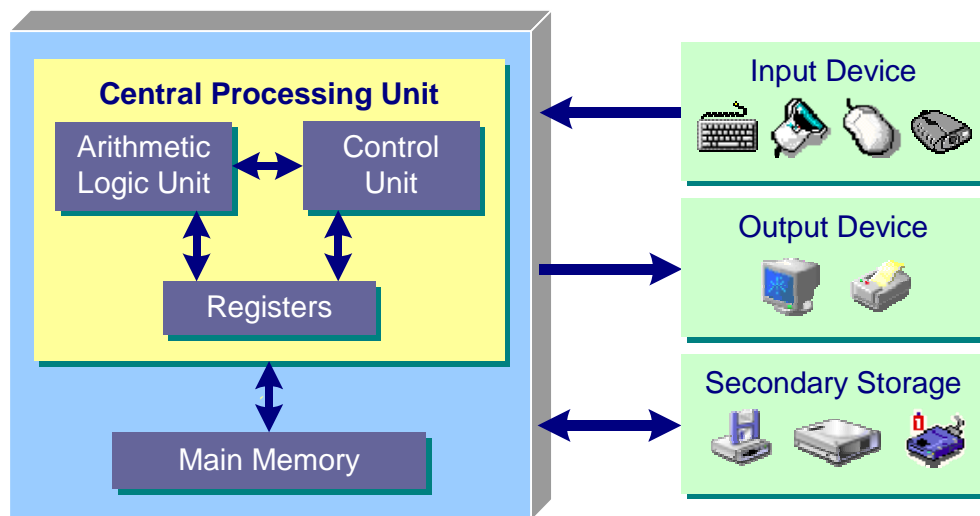


Figure 1.1 Components of a computer system.

Central Processing Unit (CPU)

CPU is the heart of a computer system. It controls and coordinates the operations of the computer system. It usually consists of an arithmetic logic unit (ALU), a control unit (CU) and registers. Registers are local storage within CPU to allow rapid access of information by CU and ALU. The control unit is responsible for regulating the activities of the CPU. It controls the flow of data and instructions from the memory, and interprets the instructions accordingly. It also communicates with the ALU to perform the required operations. The arithmetic logic unit performs fundamental arithmetic operations of addition, subtraction, multiplication and division. It also handles logical operations such as AND, OR, NOT, etc. for comparison of two data items.

Main Memory

Main memory is also called primary memory. The purpose of the main memory is

to store information that is to be processed by the CPU. Memory is grouped into *cells* or *words*. Each cell typically consists of 8-bits (or 1-byte) and has its own address. Information such as characters, numbers and instructions are stored in memory cells. The main memory comprises two parts: Random Access Memory (RAM) and Read Only Memory (ROM). RAM contains programs and data that are erased once execution finishes. ROM holds essential data and programs which are critical to the operation of a computer system. They cannot be removed even if the power is turned off. They are permanent in the memory of the computer system.

Secondary Storage

Primary storage is relatively expensive, so most computer systems have only a limited amount of primary storage. Therefore, secondary storage is needed to store information that requires to be kept for a long period of time. The three most commonly used secondary storage devices are floppy disks, hard disks and CDs. The data in the secondary storage must be transferred to the main memory first before the Central Processing Unit can process it. The secondary storage differs from the main memory as follows:

1. The access time for the secondary storage is slower than the main memory.
2. Secondary storage devices can be portable. It usually has a greater capacity than the main memory.
3. Data stored in secondary storage devices remains permanent even after a sudden loss of power.

Input and Output Devices

Input and output devices allow us to communicate with the computer system. For most computer systems, the most common input devices are keyboards, scanners, microphones and mice. The most common output devices are monitors, printers and speakers.

1.2 Computer Software

In addition to hardware, software is also required to make a computer system work to solve a problem. There are two categories of software: *system software* and *application software*. Application software refers to programs that are used to solve specific problems. System software refers to programs that are used by programmers or developers to build application software. Some examples of system software include operating systems, utility programs and language translators. This is illustrated in Figure 1.2.

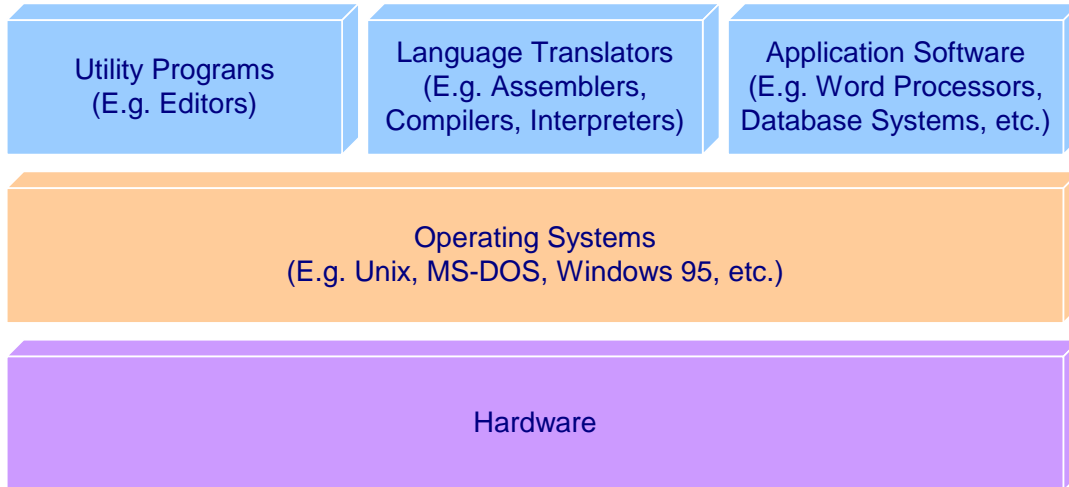


Figure 1.2 System software and application software.

Operating Systems

An operating system is responsible for managing the overall operations of a computer system. Each computer has its own operating system. Some of the well-known operating systems include the Unix operating system, the disk operating system (DOS), Windows 95 and Windows NT, and the Macintosh operating system. The major tasks performed by the operating system include user access control, process management, memory management, file management, and input and output control for various input and output devices.

Utility Programs

Utility programs are special software which are commonly included within an operating system. Examples include text editors, and special tools for file management and I/O management. A text editor is a program that enables a user to create and modify the contents of a text file. It is similar to a word processor. In some cases, it can also be considered as an application software. Almost all C programming environments provide text editors which facilitate input of C source code.

Language Translators

Language translators convert a computer program written in a high-level programming or assembly language into its equivalent machine language instructions. There are three types of language translators: *assemblers*, *compilers* and *interpreters*. Assemblers convert programs written in assembly language into

machine code, while compilers convert programs written in a high-level language into machine code. In both cases, the machine code is saved into a file before execution. Interpreters are also used for high-level languages. But it translates program instructions one at a time into machine code and immediately executes the code. The interpretation process completes when all the program instructions are translated and executed. One example of language translators is a C compiler which translates computer programs written in C into machine language.

Application Software

Application software includes word processors, database management systems, graphics applications, etc. Special software developed for business, education, government and engineering applications is also application software. This book describes the program development process on how to write application programs, especially large application programs.

1.3 Programming Languages

Programming is to design a set of instructions so that when the computer follows the instructions it will do something that the user has intended to do. A programming language is a language that is used to communicate with the computer. The instructions to the computer have to be syntactically correct. Three types of programming languages are available: Machine Language, Assembly Language and High-Level Language.

Machine Language

Machine language consists of instructions in the form of binary (or hexadecimal) code which computer systems can only understand. Since machine instructions are quite primitive, it is tedious and cumbersome to program in machine language. Therefore, other languages such as assembly language and high-level language were developed which can be translated into machine language.

Assembly Language

Assembly language replaces binary instructions by symbolic instructions in text. To use assembly language, programmers still need to understand the internal architecture of the computer system. The symbolic instructions are translated into machine instructions through a language translator called *assembler*.

High-Level Language

A high-level language allows programmers to write programs consisting of lines

of code in a form similar to English. It is relatively easier to learn to program in high-level languages. In addition, the programmer does not need to concern about the internal design of the machine on which the program is to be used. A *compiler* is a program that translates programs written in high-level language into machine language. The program written in high-level language is called *source code*. The output of the compiler is called *object code*. There are many high-level languages available now. These include C, PASCAL, BASIC, COBOL, FORTRAN, Prolog, Smalltalk, C++, etc. Each has its own special features, which are suitable for application to a certain domain. C and PASCAL are procedural languages, Prolog is a declarative language, while Smalltalk and C++ are object-oriented languages. FORTRAN has traditionally been used for engineering applications. COBOL has been widely used in mainframe systems for data processing applications that are essential in banking and commerce environments.

1.4 Why C Programming?

C programming language was created by Dennis Ritchie at AT&T Bell Laboratories in 1972. The language was originally designed as a system software development language to be used with the Unix operating system. Over the past three decades, C has become one of the most popular programming languages for developing a wide range of applications including word processors, graphics applications, database management systems, communication systems, expert systems and other engineering applications.

The C language has a number of advantages over other programming languages such as BASIC, PASCAL and FORTRAN. It is powerful, flexible, efficient, portable, structured and modular.

Powerful and Flexible

C is a powerful and flexible language. C has been used to implement the Unix operating system. Compilers for many other languages, such as FORTRAN and BASIC are also written in C. C also provides the freedom and control in developing programs. It is easy to perform operations on manipulating bits, bytes and addresses using operators and pointers. C also supports programming over hardware and peripherals.

Efficient

C is an efficient language. As C is a relatively small language, efficient executable programs can be generated from C source codes. C programs are compact in storage requirements and can be run very quickly.

Portable

A C program written in one type of computer system can be run on another type of computer system with little or no modification. As C compilers are available for a wide range of systems from personal computers to mainframes, it is easy to port C programs from one hardware platform to another.

Structured and Modular

C has features to enable the creation of well-structured programs that are easy to read. C programs can also be written in functions, which can be compiled and tested separately. Large and complex systems can be written and tested as modules by different people, these modules are then combined into a single program. The functions and modules can also be reused in other applications.

However, C also has a few disadvantages. The free style of expression in C can also make it difficult to read and understand. In addition, as C is not a strongly-typed language such as PASCAL, it is the programmer's responsibility for ensuring the correctness of the program. Pointer in C is a very useful feature, however, it can also cause programming errors that are difficult to debug and trace if it is not used properly. Nevertheless, these drawbacks can be overcome if good programming style is adopted.

1.5 ANSI C Standard

With the increasing popularity of the C language, many organizations enhanced the work of Dennis Ritchie to create their own version of C. The differences between the various C implementations cause incompatibility among each other. In 1989, the American National Standards Institute (ANSI) approved a standard definition of C, which is known as ANSI C, to resolve the incompatibility problem. The requirement of the standard is that all developers of C compilers should make their compilers capable of compiling ANSI C program code. As such, once a program is written in ANSI C, it should be able to be compiled and executed on any machines with an ANSI C compatible compiler. Currently, most C compilers support the ANSI C standard.

1.6 Structure of a C Program

A typical C program has the structure shown in Figure 1.3. Most C programs contain the following components: preprocessor instructions, global declarations, function prototypes, program statements, functions and comments.

```
Preprocessor instructions
global declarations
function prototypes

main()
{
    statements
    return 0;
}

function()
{
    statements
}
```

Figure 1.3 A typical C program structure.

Preprocessor Instructions

Preprocessor instructions refer to the instructions to the preprocessor of the compiler. All preprocessor instructions start with the symbol **#**. The C preprocessor supports string replacement, macro expansion, file inclusion and conditional compilation. For example, the **#include** *<filename>* instruction tells the preprocessor to include the file *<filename>* into the text of the source code file before compilation. The files that are to be included usually have the extension **.h** such as *stdio.h*. They are usually placed at the beginning of the source code file. The *stdio.h* file is required for programs that need to perform input/output operations.

Global Declaration

Global declaration statements declare global variables that are accessible anywhere within a C program.

Function Prototypes

Function prototypes provide useful information regarding the functions used in the program to the C compiler. They inform the compiler about the name and arguments of the functions contained in the program. They need to appear before the functions are used.

Functions

A C program consists of one or more functions. The **main()** function is required

in every C program. A function has a header and a body. The header is **main()**. The body of the **main()** function is enclosed by the braces **{}**. It consists of statements or instructions that the computer can execute. Program execution starts from the beginning of the body. The statement

```
return 0
```

can be the last statement of the **main()** function depending on whether the function returns any values. The statement

```
void main()
```

does not require a **return** statement in the **main()** body. The statement

```
int main()
```

requires the **main()** body to have a **return** statement to return an integer value. The format of the statement

```
main()
```

is also accepted by ANSI C, which is equivalent to **int main()**.

Apart from the **main()** function, a program can have many other functions. The function header defines the name of the function, and the inputs expected by the function. The function body contains statements that are written to perform a specific task. The body will be executed when the function is called. The **main()** function or functions can call other functions, which may in turn call other functions.

A function may or may not have arguments, and it may or may not have a return value. The function

```
int add(int x, int y)
```

has two arguments and requires a return value, while the function

```
void add()
```

has no argument and does not require a return value.

In addition, C provides library functions that perform most of the common tasks such as input operation from the keyboard and output operation to the screen. The **printf()** and **scanf()** statements are the most commonly used library functions for input/output operations. The **printf()** statement displays

information on the screen, while the `scanf()` statement reads data from the keyboard and assigns the data to a variable.

Program Statements

A statement is a command to the computer. There are two types of statements: *simple* or *compound* statements. A simple statement is a statement terminated by a semicolon (;). There are four types of simple statements: declaration statements, assignment statements, function call statements and control statements. Some examples of C statements are given as follows:

```
int x,y,z;           /* declaration statement */
x = 30;              /* assignment statement */
printf("Welcome to Introduction to C Programming");
                    /* function call statement */
for (y=0; y<z; y++) {...} /* control statement */
```

A compound statement is a sequence of one or more statements enclosed in braces:

```
{
    statement_1;
    statement_2;
    ...
    statement_n;
}
```

Each of the `statement_i` (where `i = 1..n`) can be a simple or a compound statement.

Comments

Comments may be added to a program to explain what is the purpose of the program, or how a portion of the program works. A comment is a piece of English text. It is enclosed by `/*` and `*/`. Comments may appear anywhere in the program. It makes programs more readable. The compiler simply skips the comments when translating the program. There are different ways to write comments:

```
/* This is a comment */

int x,y,z; /* This is another comment */

/* This is the first comment
This is the second comment
```

```
This is the third comment */
```

It is important to have comments in the programs. Comments document what the program does, how the variables are defined, and how the logic of the program works. This is extremely helpful when future modification of the program is required. Therefore, it is necessary to develop the habit of using comments to document the programs including the use of comments for programming structures and operations.

1.7 A Sample C Program

A sample C program is shown in Program 1.1 that computes the summation of two input numbers. The first line is the comment, which documents the purpose of the program. The **#include** preprocessor directive instructs the C compiler to add the contents of the file *stdio.h* into the program during compilation. The *stdio.h* file is part of the standard library. It supports input and output operations that the program requires.

The next statement

```
int add(int, int);
```

is a function prototype that informs the compiler on the type and number of arguments that the function expects, and the return data type. In function **add()**, it expects two arguments of type **int**, and will return a value of type **int**.

In the **main()** function, it requires to return an integer value, so the keyword **int** is used to inform the C compiler. The braces **{}** are used to enclose the **main()** function body.

The statement

```
int x,y,z;
```

is the declaration statement that informs the compiler the creation of three variables of data type **int**. The variables will be used to store integer numbers. Memory storage space will be allocated to the three variables when this statement is executed.

The **printf()** function is the library output function call to print a character string on the screen. The **scanf()** function is the library input function to read in two integer values, and store the values in the variables **x** and **y**.

The function call

```
z = add(x,y);
```

calls the function `add()` using the argument values from variables `x` and `y`. The function will then be executed, and the values of `x` and `y` are passed into the function and assigned to integer variables `a` and `b`. The `return` statement of the function `add()` then performs the addition operation of the variables `a` and `b` and returns the result back to the calling `main()` function. The result will be assigned to the variable `z` and stored in the memory location of `z`.

The next statement after the function call will then be executed, which is the last `printf()` statement. It prints out the summation result stored in the variable `z` of data type `int`. Finally, the statement `return 0` returns the control back to the system.

Program 1.1 A sample program structure.

```
/* A sample program to perform addition of two numbers */
#include <stdio.h>      /* preprocessor directive */
int add(int, int);     /* function prototype */

main(void)             /* the main() function header */
{                       /* begin of main() function body */
    int x,y,z;         /* variable declaration */

    printf("Welcome to Introduction to C Programming.\n");
                        /* call printf() library function */
    printf("Enter the first number: ");
    scanf("%d", &x);    /* call scanf() library function */
    printf("Enter the second number: ");
    scanf("%d", &y);
    z = add(x, y);      /* assignment statement and function
                        call */
    printf("%d + %d = %d", x, y, z);
    return 0;
}                       /* end of main() function body */

int add(int a, int b)  /* function definition */
{
    return (a + b);     /* return statement */
}
```

Program input and output

```
Welcome to Introduction to C Programming.
Enter the first number: 24
Enter the second number: 30
24 + 30 = 54
```

1.8 Development of a C Program

To develop a C program, a text editor is used to create a file containing the source code in C. Then, the source code needs to be processed by a *compiler* to generate an object file. The *linker* is then used to link all the object files to create an executable file. Finally, the executable file can be run and tested. Figure 1.4 shows the steps involved in creating a C program. Programs can be developed and executed on different environments such as the Unix system, Microsoft Windows or MS-DOS environment. In the following sections, we focus the discussion on developing C programs under the Unix system as C began from there.

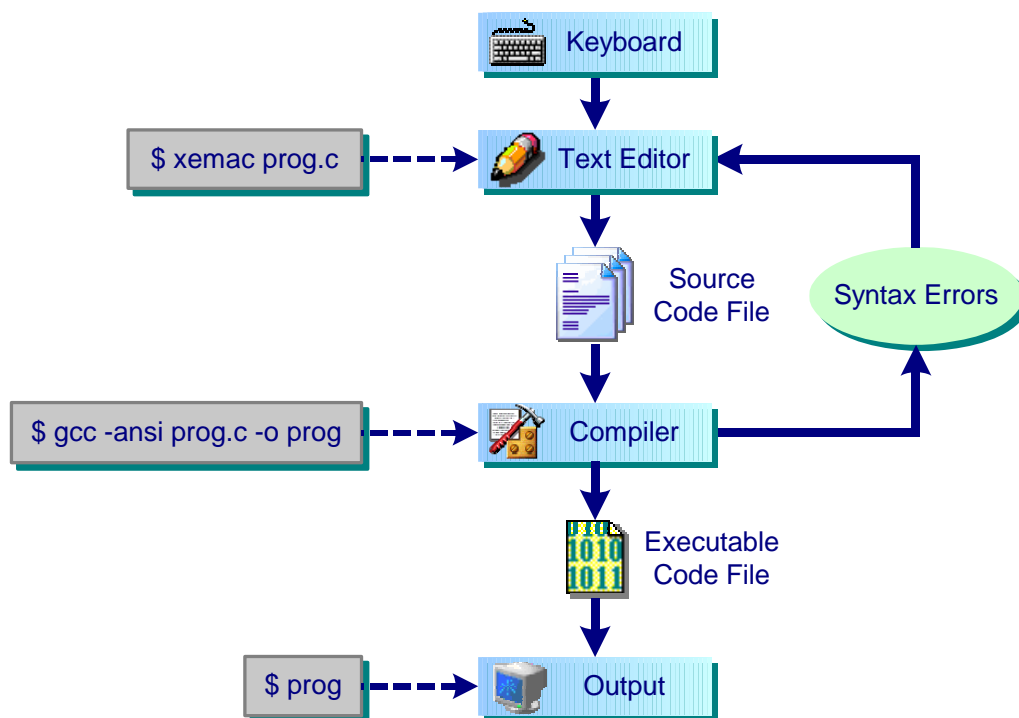


Figure 1.4 Steps in developing a C program.

Creating C Source Code Files

The first step in the development of a C program is to enter a source code into an editor. Most compilers come with editors that can be used to enter and edit source code. If it is not available, editors available in Unix systems such as *ed*, *ex*, *edit*, *emacs* and *vi* can be used instead.

Program 1.2 and Program 1.3 show two sample C programs that are created using an editor. The source code file in Program 1.2 is saved as **welcome.c**, and the source code file in Program 1.3 is saved as **sqroot.c**. The name given to the

source file should be meaningful that describes the purpose of the program. The file extension (`.c`) is used for C program source files.

Program 1.2 Sample program welcome.c.

```
/* Print a welcome message on the screen */
#include <stdio.h>

main(void)
{
    printf("Welcome to Introduction to C Programming.\n");
    return 0;
}
```

Program output

Welcome to Introduction to C Programming.

Program 1.3 Sample program sqroot.c.

```
/* Print the square root of 2.0 on the screen */
#include <stdio.h>
#include <math.h>

main(void)
{
    printf("The square root of 2 is %f.\n", sqrt(2.0));
    return 0;
}
```

Program output

The square root of 2 is 1.414214.

Compiling and Linking C Programs

The compilation and linking processes are illustrated in Figure 1.5. The *compiler* checks for syntax errors in the source files. Error messages will be given if errors are found. The locations of the errors are also given in the error messages. All the errors need to be corrected. If no more errors are detected, the compiler then translates the source code into machine language instructions, which are called object code. A file is created to store the object code. The object file has the same name as the source file, but with a different extension (`.obj` or `.o`). The extension

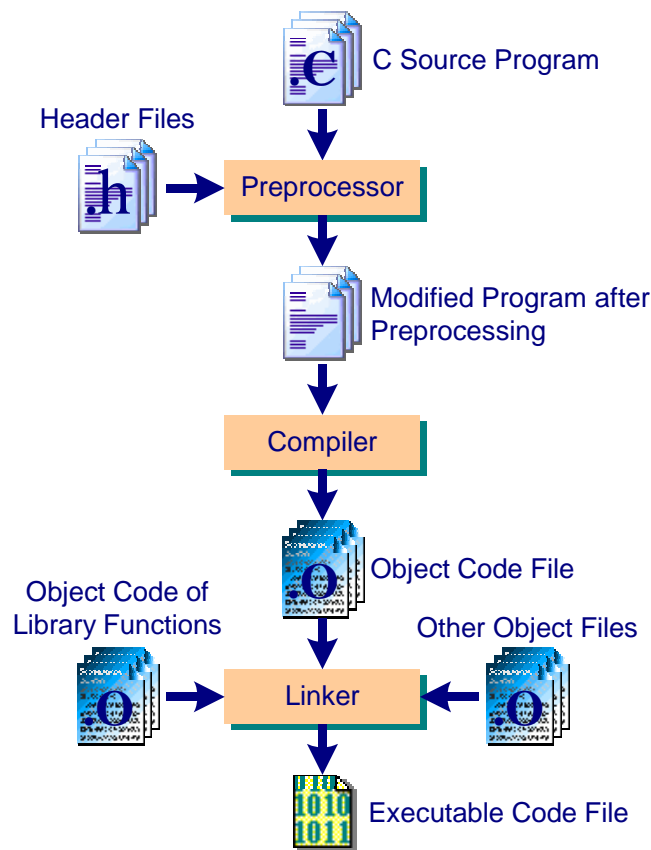


Figure 1.5 Preprocessor, compiler and linker.

is used to indicate that the file contains object code.

After compilation, the *linker* combines the object code with other object files to produce an executable program. The linker allows us to develop programs that can be placed in several files. Functions that are frequently used in other programs are placed in separate files and compiled separately. This can save time to re-compile the functions every time they are used. In addition, only files containing updated source code need to be re-compiled before linking. The linker also provides a convenient way to support the use of standard libraries. Most C programs use standard library functions to perform certain specific tasks such as input/output operations and mathematical functions. During linking, the linker combines the object code with the object code from the standard library provided by the compiler to generate the executable program.

The commands that are used to invoke the compiler depend on the program development environment. In most environments, the compilation and linking processes are integrated together. The compiler will invoke the linker after the compilation process if no errors are found. The Unix C compiler and Microsoft C compiler are examples of compilers that integrate the compilation and linking

processes.

To illustrate the compilation process, a compiler called **gcc** (GNU C Compiler Collection) which is a free software available at <http://gcc.gnu.org/> is used to compile C programs that run in the Unix environment. We can type the following command to compile the **welcome.c** program:

```
$ gcc -ansi welcome.c
```

where the file **welcome.c** contains the program. **gcc** is the command to call the compiler. **-ansi** is a compiler option to compile the program according to the ANSI C standard. If the program has no errors, the compiler will call the linker automatically to do the linking and produce the executable file called **a.out**. The object file created will be **welcome.o**. The object file will be removed once the executable program file is created. In the case when more than one source code file are used during compilation and linking, the object code files are saved and retained.

We can also compile the program and give a name to the executable file instead of **a.out**. To do this, we may type

```
$ gcc -ansi welcome.c -o welcome
```

The **-o** option tells the linker to name the executable file **welcome** instead of the default filename **a.out**.

As shown in **sqroot.c**, the program uses the library function **sqrt()**, from the *math* library to compute the square root of a number. We need to inform the compiler the library we use. The compilation command becomes

```
$ gcc -ansi sqroot.c -o sqroot -lm
```

The **-l** option is used to tell the compiler the library in which the special function used in the program is found. **m** indicates the *math* library. In addition to the change in the **gcc** command, we also need to add the statement

```
#include <math.h>
```

at the beginning of the program in **sqroot.c** to tell the preprocessor to include the definition file of the *math* library.

Executing a C Program

After the compilation and linking processes, an executable program is created. To run the program, we can type

```
$ a.out
```

or if we have given a name to the executable file, e.g. **welcome** or **sqroot**, then we can just type

```
welcome  
or sqroot
```

Then the program will be executed.

1.9 Exercises

1. Describe the four basic hardware components of a computer system.
2. Explain the difference between system software and application software.
3. Explain the difference between machine language, assembly language and high-level language.
4. List five high-level programming languages.
5. List the strengths and weaknesses of the C language.
6. Describe the processes involved to generate an executable code when the following command is typed on a workstation in ANSI C mode:

```
gcc -ansi prog.c -o prog
```

State the function, input and output of each process.



Program Development

We write a program to solve a particular problem. To do this, it is necessary to have a thorough understanding of the problem, and a structured approach for the development of the program. This chapter discusses the techniques for the development of structured programs. In particular, we discuss the top-down stepwise refinement technique as an approach to developing programs for problem-solving.

This chapter covers the following topics:

- 2.1 Program Development Process
 - 2.2 Program Design
 - 2.3 Flowcharts and Pseudocode
 - 2.4 Control Structures
 - 2.5 Top-down Stepwise Refinement
 - 2.6 Implementation
 - 2.7 Program Testing
 - 2.8 Documentation
-

2.1 Program Development Process

To develop a program to solve a problem, the program development process generally consists of 6 steps as shown in Figure 2.1. They are Problem Definition, Problem Analysis, Program Design, Implementation, Program Testing and Documentation.

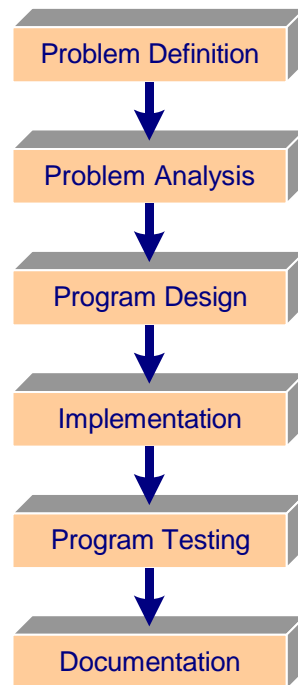


Figure 2.1 Program development process.

Step 1: Problem Definition

This step determines the objective of the program, and writes a problem statement or paragraph describing the purpose of the program. The problem statement is a broad statement of the requirements of the program, in user terms.

Step 2: Problem Analysis

This step analyzes the problem and produces a set of clear statements about the way the program is to work. This requires a clear understanding of the underlying concepts and principles of the problem. These statements define how the user uses the program (*program input*), what output the program will generate (*program output*), and the *functionality* of the program. The functionality may be expressed in terms of mathematical formulas or equations for specifying the transformation from input to output.

Step 3: Program Design

This step is to formulate the program logic or *algorithm*. An algorithm is a series of actions in a specific order for solving a problem. There are two basic methods that can be used to design the program logic: *pseudocode* and *flowcharts*. The technique that is used for developing a program is the top-down stepwise refinement technique.

Step 4: Implementation

This step is to convert the program logic into C statements forming the program. If the program has been designed properly, then it is a straightforward task to map the program design into the corresponding C code.

Step 5: Program Testing

This step is to test the program code by running the program. This aims to determine whether the program does carry out the intended functionality. Program testing involves the use of test data that the correct answers are known beforehand, and the use of different test data to test the different computational paths of a program. Therefore, it is important to design test cases such that all conditions that can occur in program inputs are tested. However, it is also necessary to ensure that the tests are not too exhaustive.

Step 6: Documentation

This step is to document the programs. Documentation of computer code is useful for the understanding of the program's design and logic. This is important for the maintenance and future modification of the programs. In addition, documentation such as user manuals can also help users to understand on how to use the program.

2.2 Program Design

Program design is one of the most important steps in the program development process. It aims to design the logic of the program or algorithm. When we write an algorithm for solving a given problem, we need to design our algorithm in the following ways. First, an algorithm must be unambiguous. Every step of the algorithm must be clear and precise. An algorithm must also specify the order of steps precisely. We need to consider all possible points of decision, and for each decision point, we need to consider a step for each possible outcome. Finally, the algorithm must execute the steps and terminate in finite time.

Consider an example that determines the largest number from a series of numbers entered by a user. To design an algorithm for this problem, we can develop the algorithm comprising the following steps:

1. Initialize the variable **max** to 0.
2. Read the number of data items **n** from the user.
3. For each input number, compare the variable **number** that stores the input number with the variable **max**, determine the maximum value and update the value into **max**. The process stops when the number of user inputs **n** has been reached.
4. Print the maximum value **max**.

2.3 Flowcharts and Pseudocode

Program logic may be represented by flowcharts or pseudocode. Both can be used in the design of a program. A flowchart is a way to represent the program logic or algorithm by a diagram. It describes the logic of the operations and the sequence in which they are carried out by the program. Pseudocode uses English-like statements to design and describe the operations performed by the program. Both flowcharts and pseudocode are useful tools to help us determining the order of the operations to solve a problem. They are also good tools for documentation. It is easy to develop programs directly from flowcharts or pseudocode. In this section, we describe both flowcharts and pseudocode.

Flowcharts

In flowcharts, the flow of the control can be easily visualized. A flowchart is composed of a set of standard symbols. The symbols are connected by *flowlines*. Figure 2.2 shows some of the common symbols used in constructing flowcharts.

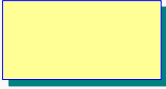
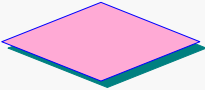
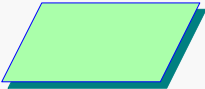
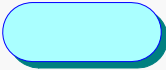
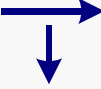
Symbol	Name
	Process
	Decision
	Input / Output
	Terminal
	Flowlines

Figure 2.2 Flowchart symbols.

- **Process Symbol.** The process symbol, which is a rectangle symbol, is used to indicate any types of processing or calculation. Process symbols are normally used to represent a collection of statements that perform the calculation.

- **Decision Symbol.** The decision symbol, which is a diamond symbol, is used to indicate that a decision is to be made at a certain point in a flowchart at which a branch to one or more alternative paths is possible. It contains a condition that can either be true or false. Two flowlines leaving the corners of the diamond are labeled based on the condition in the symbol. One indicates the direction to be taken when the condition is true, and the other indicates the direction to be taken when the condition is false.
- **Input/Output Symbol.** The input/output (I/O) symbol, which is represented by the parallelogram symbol, is used to indicate the input and output operations. It corresponds to the operations performed by I/O functions such as the `scanf()` and `printf()` statements.
- **Terminal Symbol.** The terminal symbol, which is oval-shaped, is used to indicate the beginning or end of a program. It can also be used to specify the beginning or end of a function. The terminal symbol containing the word "Start" is the first symbol used in a flowchart in a program; and the terminal symbol containing the word "End" is the last symbol used.
- **Flowlines.** The flowlines are used to connect symbols and indicate the direction of processing.

Pseudocode

Another method to describe the program logic is pseudocode. Pseudocode is an informal language. There are no strict rules in writing pseudocode for program logic. It uses a mix of English, together with a set of special keywords to describe the operations of a program. Examples of some of the commonly used keywords are *begin*, *end*, *if*, *else*, *else_if*, *end_if*, *while*, *do*, *end_while* and *end_do*. The keywords *read* and *print* are used for input and output. Verbs such as *set*, *initialize*, *compute*, *add* and *subtract* can also be used depending on the nature of the statements.

2.4 Control Structures

A structured program can be written using simple control structures to solve a problem. The basic control structures are sequence structure, selection (or branching) structure and repetition (or looping) structure. Structured programs should employ only these three basic control structures. Bad control structures such as the `goto` statement should be avoided. Structured programs are easier to debug and modify. In addition, they are also easy to be read and understood by other programmers.

Sequence Structure

A sequence structure contains a sequence of statements (or operations) that are performed one after another. The pseudocode and flowchart representations of a sequence structure are shown in Figure 2.3. In the pseudocode, we use the keywords **begin** and **end** to define the beginning and end points of the structure. The operation performed in each statement may be computation, input or output operation.

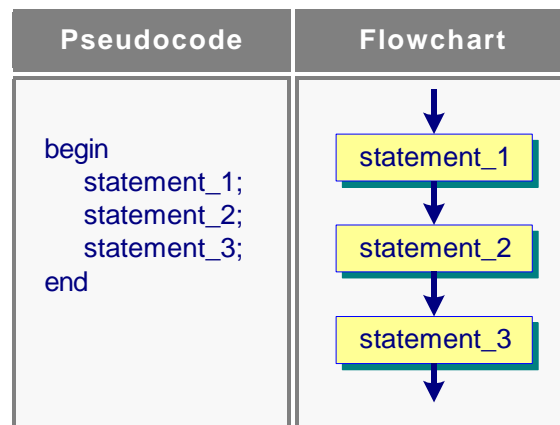


Figure 2.3 Sequence structure.

Selection (Branching) Structure

A selection structure contains a set of statements that is performed if a *condition* is true, and another set of statements that is performed if the condition is false. The **if** structure is the most commonly used structure to describe selection. The pseudocode and flowchart for the simple **if** structure and **if-else** structure are shown in Figure 2.4. *statement*, *statement_1* and *statement_2* may be a single statement or a series of statements. In the **if-else** structure, if the condition is true, then *statement_1* will be executed. If *statement_1* is a series of statements, it must be enclosed inside a block using the keywords **begin** and **end** when using pseudocode. If the condition is false, then *statement_2* will be executed.

Repetition (Looping) Structure

A repetition structure contains a set of statements that is repeated as long as a condition is true. In the **while** loop structure, a test *condition* is used to control the number of times the loop is going to repeat. The loop will be executed until the condition is satisfied. The **do-while** loop structure differs from the **while** loop

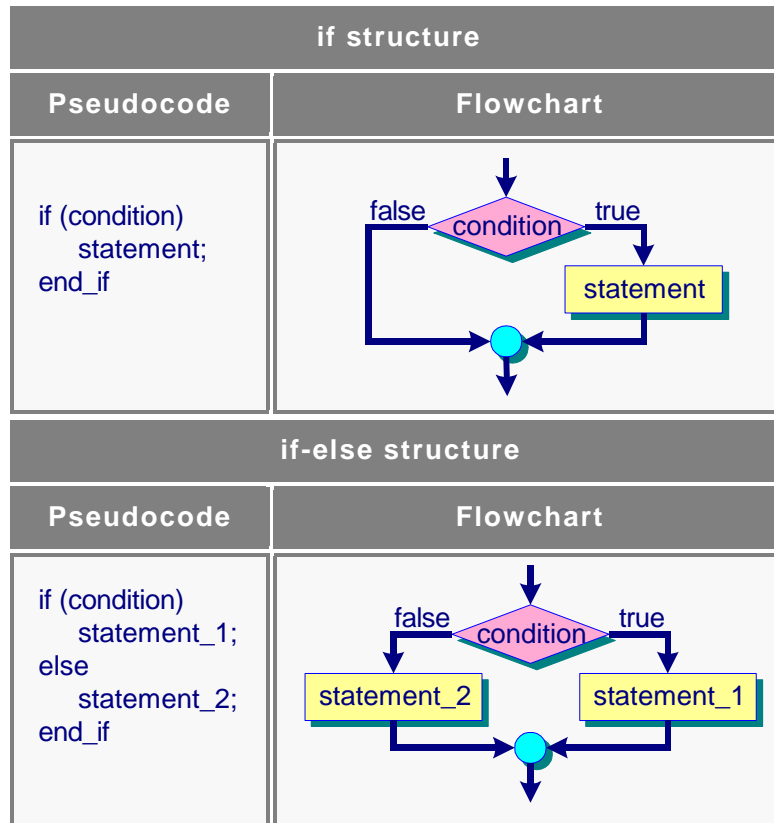


Figure 2.4 Selection structure.

structure in which the test condition is evaluated after a set of statements that is to be repeated has been executed. This set of statements will be executed at least once no matter whether the test condition is satisfied or not. The pseudocode and flowchart for the **while** loop structure and **do-while** structure are shown in Figure 2.5.

Example

The following example illustrates how to use pseudocode and flowchart for the logic design of the algorithm discussed in Section 2.2.

Problem Definition: Write an algorithm to read in a series of numbers entered by a user, compute the largest number, and print the number to the screen.

Pseudocode Design: The pseudocode of the algorithm is given as follows:

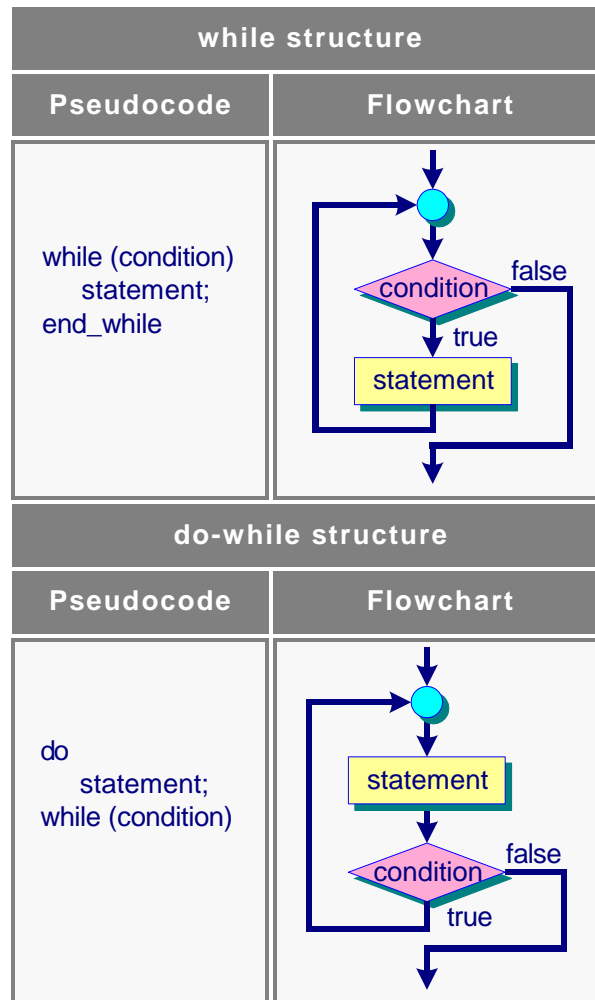


Figure 2.5 Repetition structure.

```

begin
  max ← 0
  count ← 0
  read n
  while count < n
    read number
    if max < number
      max ← number
    end_if
    count ← count+1
  end_while
  print max
end
```

The arrow (\leftarrow) is used to assign the value from the right hand side to the left hand side. This is an assignment operation.

Flowchart Design: The flowchart representation is given in Figure 2.6.

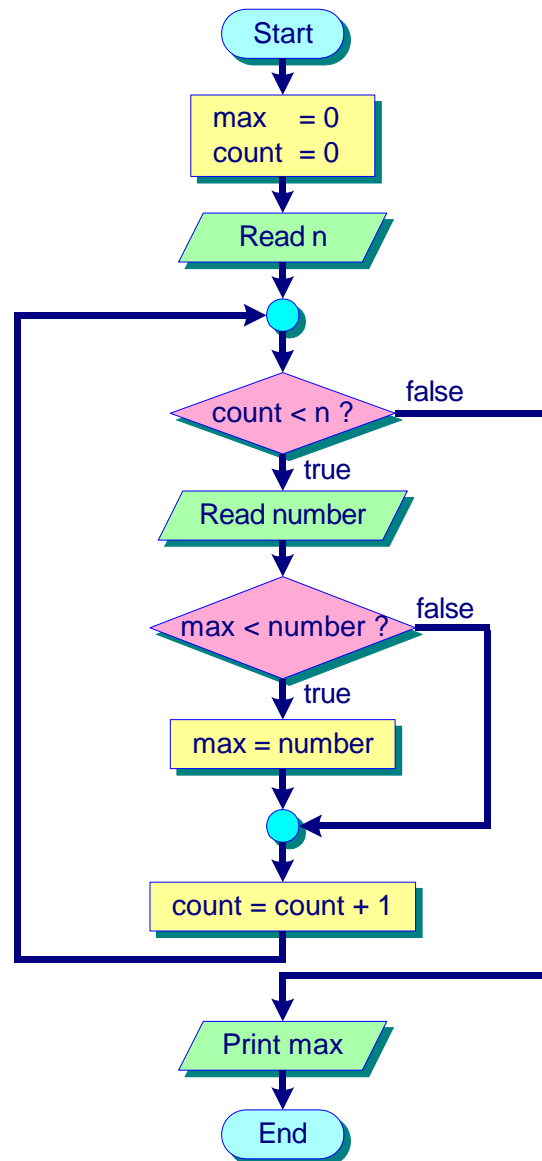


Figure 2.6 Flowchart design.

2.5 Top-down Stepwise Refinement

The top-down stepwise refinement technique uses the divide-and-conquer strategy. It consists of two techniques: *decomposition* and *stepwise refinement*. Decomposition splits a large problem into a series of smaller subproblems. Then, consider each subproblem separately and further split them into subproblems. For stepwise refinement, we refine the steps of the problem solutions to each subproblem in greater detail until no further refinement is possible. The refinement process stops when all steps are detailed enough so that they can be directly translated into a C program. Finally, all the final refinements of subproblems are combined according to the logic of the original problem. Figure 2.7 illustrates the top-down stepwise refinement technique for problem solving.

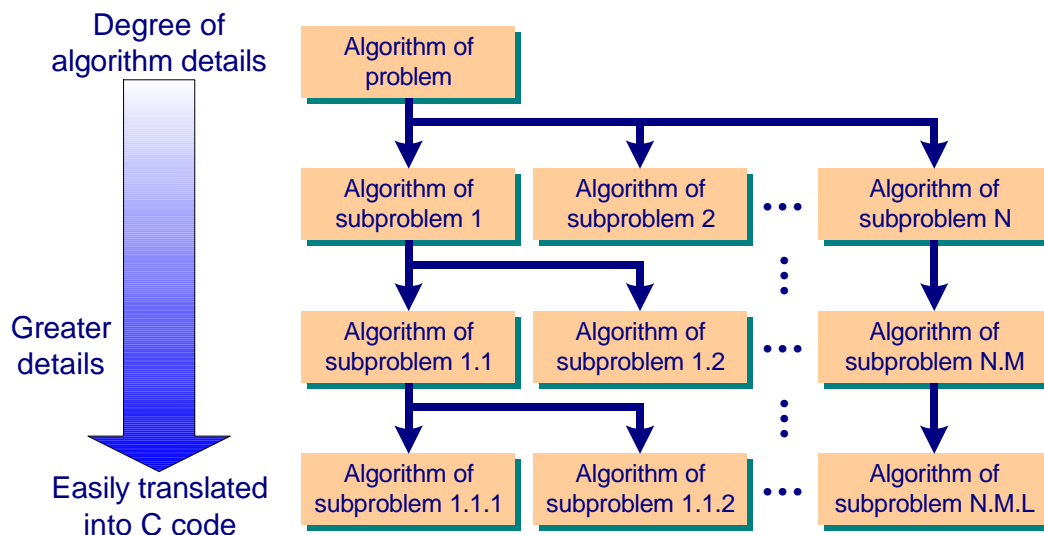


Figure 2.7 Problem solving using top-down stepwise refinement.

Example

Consider the following problem definition:

Problem Definition: All students from the College of Engineering, NTU can take the Introduction to C Programming course in their first year of study. The students are required to take an examination at the end of the course. In order to know how the students perform during the examination, you are asked to write a program to generate the statistics of the examination results. You will be given a list of marks for all the students who have taken the course in a specified semester and year. The program should find the average mark, the number and percentage of passes

and failures. The passing mark is 50.

Problem Analysis: A *context diagram*, which is a complete representation of the program, can be used as the first step on problem analysis. The purpose, and the required input and output of the program are stated in the diagram. In addition, we can express the relationships between the input and output through mathematical equations. If constraints or special conditions need to be considered in the program, they should also be stated during problem analysis. For the above problem definition, we can first derive the context diagram as shown in Figure 2.8, and then write down the purpose of the program, and the required inputs and outputs of the program.

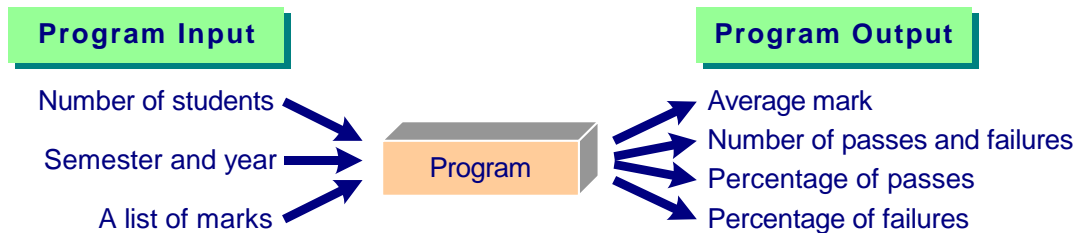


Figure 2.8 Context diagram.

Purpose: For a specified semester and year, process a list of student marks to generate the average mark, the number of passes and failures, and the percentage of the number of students who passes or fails the course. The passing mark is 50.

Required inputs:

- (1) the semester and year
- (2) the number of students
- (3) a list of marks

Required outputs:

- (1) the average mark
- (2) the number of passes and failures
- (3) the percentage of passes and failures

Formulas:

- (1) average mark = total marks / number of students
- (2) percentage of passes = total number of passes / number of students
- (3) percentage of failures = total number of failures / number of students

Program Design: The whole problem can be accomplished by carrying out the three subtasks in the flowchart shown in Figure 2.9. The context diagram has thus been refined into the flowchart. Three subtasks are defined:

1. Read and initialize variables
2. Input the marks, and process the marks to generate the statistics
3. Print the statistics

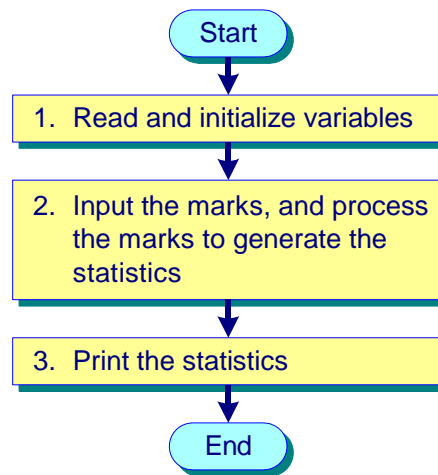


Figure 2.9 Flowchart for the program.

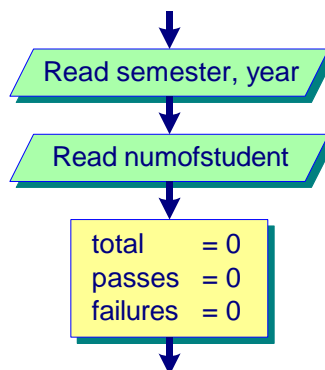


Figure 2.10 Flowchart for subtask 1.

The flowchart is also a complete representation of the program, but further refinement is necessary for each subtask of the program. The flowchart for subtask 1, *Read and initialize variables* is given in Figure 2.10. In the flowchart, the

variable **numofstudent** is defined as the number of students. Subtask 1 is refined as:

- 1.1 Read semester, year
- 1.2 Read the number of students, numofstudent
- 1.3 Initialize total=0, passes=0, failures=0

The flowchart for subtask 2, *Input the marks, and process the marks to generate the statistics* may be refined as shown in Figure 2.11. In subtask 2, three subtasks are further defined:

- 2.1 Input a student's mark
- 2.2 Add the mark to totalmark
- 2.3 Update passes or failures

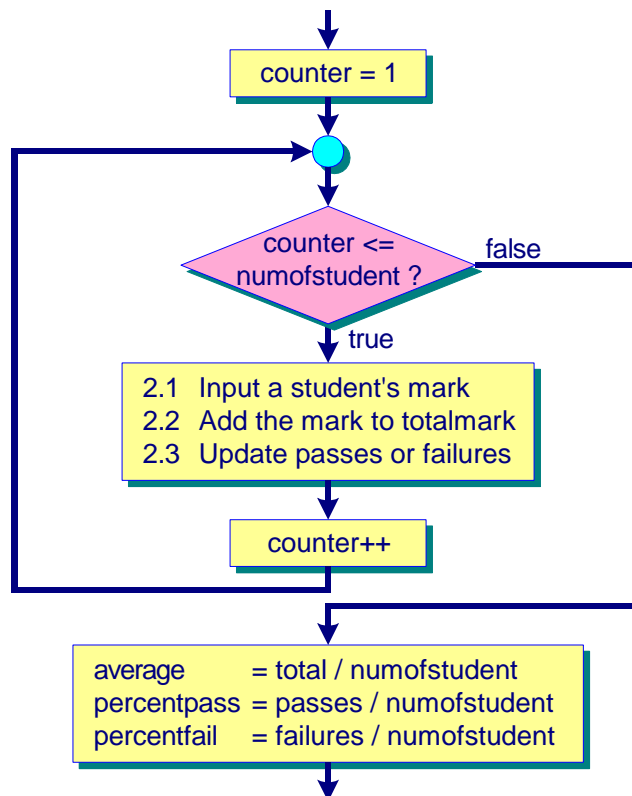


Figure 2.11 Flowchart for subtask 2.

The subtask 3, *Print the statistics* may be refined as:

- 3.1 Print semester and year
- 3.2 Print average
- 3.3 Print passes

- 3.4 Print failures
- 3.5 Print percentage of passes
- 3.6 Print percentage of failures

Figure 2.12 shows the flowchart for subtask 3.

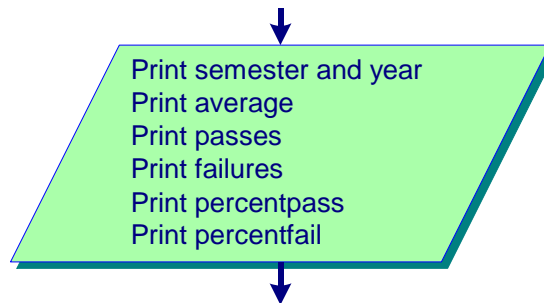


Figure 2.12 Flowchart for subtask 3.

Furthermore, the subtask 2.3, *Update passes or failures* may be refined as shown in Figure 2.13. A condition is used to test whether the student's mark is less than 50. If the mark is less than 50, the variable **passes** will be increased by 1; otherwise the variable **failures** will be increased by 1.

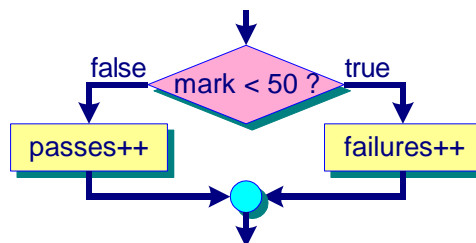


Figure 2.13 Flowchart for subtask 2.3.

Finally, Figure 2.14 shows the flowchart that combines the logic of all the subtasks.

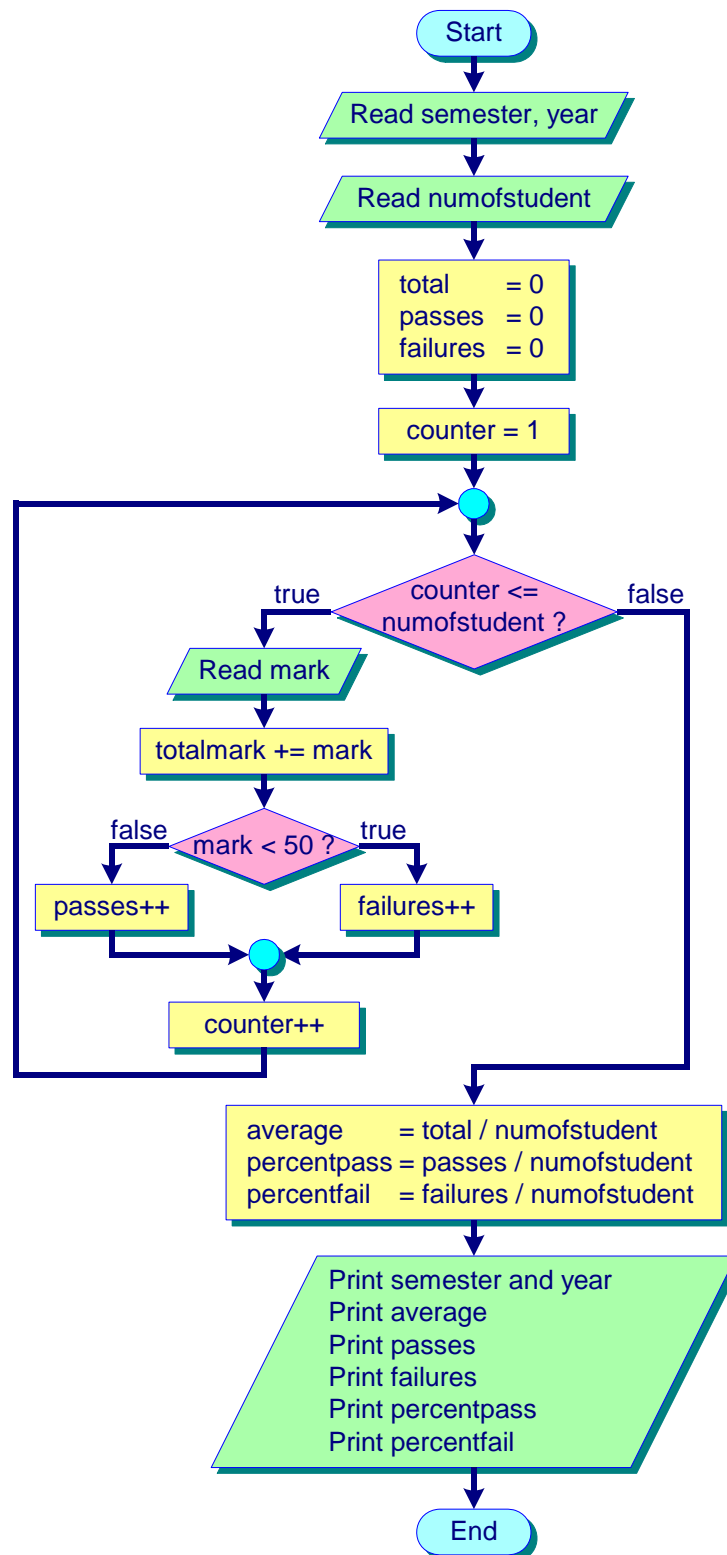


Figure 2.14 Overall flowchart.

2.6 Implementation

The next step in program development is the implementation of the algorithm in C. This can be done by translating the flowchart into program statements in C. The program is shown in Program 2.1.

Program 2.1 Program implementation.

```
/* Purpose: For a specified semester and year, process a
list of student marks to generate the average mark, the
number and percentage of passes and failures.
Author: Student name
Date: 1 April 2002
*/
#include <stdio.h>
main(void)
{
    float average, percentpass, percentfail;
    int semester, year;
    int numofstudent, mark, totalmark, passes, failures;
    int counter;

    /* initialization */
    printf("Enter the semester and year: ");
    scanf("%d %d", &semester, &year);
    printf("Enter the number of students: ");
    scanf("%d", &numofstudent);
    totalmark = passes = failures = 0;

    /* input and process the marks */
    for (counter = 1; counter <= numofstudent; counter++) {
        printf("Enter mark: ");
        scanf("%d", &mark);
        totalmark += mark;
        if (mark < 50)
            failures++;
        else
            passes++;
    }
    average = (float)totalmark / (float)numofstudent;
    percentpass = (float)passes / (float)numofstudent;
    percentfail = (float)failures / (float)numofstudent;

    /* output the statistics */
    printf("For semester %d and year %d \n", semester, year);
```

```
printf("The average mark = %f\n", average);
printf("The number of passes = %d\n", passes);
printf("The number of failures = %d\n", failures);
printf("The passing rate = %d%%\n", (int)(percentpass *
    100));
printf("The failure rate = %d%%\n", (int)(percentfail *
    100));
return 0;
}
```

Program input and output

```
Enter the semester and year: 1 2002
Enter the number of students: 10
Enter mark: 67
Enter mark: 89
Enter mark: 78
Enter mark: 35
Enter mark: 52
Enter mark: 29
Enter mark: 89
Enter mark: 76
Enter mark: 68
Enter mark: 90
For semester 1 and year 2002
The average mark = 67.300003
The number of passes = 8
The number of failures = 2
The passing rate = 80%
The failure rate = 20%
```

Programming Errors

Programs are bound to contain errors when they are first written. There are mainly three types of programming errors: *syntax errors*, *run-time errors* and *logic errors*.

- **Syntax errors.** These errors are due to violations of syntax rules of the programming language. They are detected by the compiler during the program compilation process. They are also called compilation errors. The compiler will generate diagnostic error messages to inform the programmer about the locations in the program where the errors have occurred. There are two types of diagnostic messages: warning diagnostic messages and error diagnostic messages. A warning diagnostic message indicates a minor error that might cause problems in program execution. However, the compilation is not terminated. Error diagnostic messages are serious syntax errors, which

stop the compilation process. The nature and locations of the errors are given in the messages. If the programmer cannot locate the error according to the location given by the compiler error message, then the programmer should also look at the statements preceding the stated error. If this is unsuccessful, the programmer should also check all the statements related to the stated error statement. Examples of this type of errors include illegal variable names, unmatched parentheses, undefined variable names, etc.

- **Run-time errors.** These errors are detected during the execution of the program. They are caused by incorrect instructions that perform illegal operations on the computer. Examples of such illegal operations include division by zero, storing an inappropriate data value, etc. When run-time errors are detected, run-time error messages are generated and program execution is terminated.
- **Logic errors.** These errors occur due to incorrect design of the algorithm to the problem. Such errors are usually difficult to detect and correct since no error messages are given out. Debugging of logic errors requires a thorough review of the program development process on problem analysis, algorithm design and implementation. Tracing of program logic and testing of individual modules of the program are some of the techniques, which can be used to fix logic errors.

Program Debugging

Since errors can occur during program development, program debugging is necessary to locate and correct errors. A number of techniques are available for program debugging. The most common approach is to trace the progress of the program. This approach is especially useful for debugging logic errors. Other techniques such as isolation of sections of program code and hand simulation can also be used to debug program errors.

Program tracing is to write code during program development to produce extra printout statements to monitor the progress of a program during program execution. This can give us an idea how the program works during the course of execution. Intermediate results can be printed and we can check to see whether the results are the intended ones. It is quite frustrating when a program runs but nothing is on the screen. The program could be running in a loop repeatedly, or it could be just waiting for user input. If printout statements are displayed on the screen, we should then know what the program is currently doing.

To do this, debugging printout statements such as `printf()` may be placed at several strategic locations inside the program. They can be used to print the input data read from the user, or to print computation results at different stages during program execution. Debugging printout statements are also very useful for printing data values inside a loop to show changes of these values in the loop. In addition,

printout statements may also be inserted at the beginning of each function, so that the exact execution path of the program can be traced.

Program 2.2 shows how to add debugging statements in the C program that was shown in Program 2.1. In this program, debugging printout statements have been placed at two locations. The first one is located just after reading the input value to ensure that the input value read by the program is correct. The second is located inside the **for** loop. At the end of each execution of the **for** loop, the values for the key variables **mark**, **failures** and **passes** are displayed. Thus, we can trace the progress of the program.

Program 2.2 Adding debugging printout statements into a C program.

```
#include <stdio.h>
main(void)
{
    ....
    scanf("%d", &numofstudent);
    total = passes = failures = 0;
    printf("student is: %d\n", student);
                                   /*debugging statement */

    for (counter = 1; counter <= numofstudent; counter++) {
        scanf("%d", &mark);
        totalmark += mark;
        if (mark < 50)
            failures++;
        else
            passes++;
        printf("mark: %d, failures: %d, passes: %d\n", mark,
            failures, passes);      /* debugging statement */
    }
    ....
    return 0;
}
```

Apart from tracing the program, program code can also be separated into different sections, and then each section can be tested separately on its correctness. Once the errors are isolated, you may correct the errors or rewrite the code for the erroneous section. Hand simulation can also be used by pretending that we run the program in sequence on the computer. All the values for variables and program outputs are recorded. However, this method is only suitable for small programs.

In addition, there are Unix debuggers such as GNU DDD and GDB (available at <http://www.gnu.org/software/ddd/>) which enable programmers to inspect a program during its execution. A debugger can run with a program. We

can interact with the debugger to interpret the program instructions in a step-by-step manner. We may also see the results of computations, and monitors the progress of the program. GDB provides a command-line interface, while DDD has a graphical user interface. As each compiler's debugger is different, it is necessary to consult the compiler's documentation on how to use it. Learning how to use the debugger will help to speed up your program development process.

2.7 Program Testing

Program testing is a process of executing a program to check its correctness. A program is correct if it returns the correct result for every possible combination of input values. Exhaustive testing uses all possible combinations of input values and checks whether the output is correct. However, this will sometimes take a very long time to show whether the program is correct, which is impractical except for small programs. Therefore, it is essential to have effective testing done systematically. The most important consideration in testing is the selection of test data. The data should be selected that causes every program path to be executed at least once. However, it is impractical to test all possible flow paths if a program is large.

Consider the following segment of C code for the **if** statement:

```
scanf("%d", &number);  
if (number < 10)  
    statement_1  
else  
    statement_2;
```

In this case, the boundary data such as 10 should be considered. There are two sets of test data. The variable **number** may take a value less than 10, and a value that is equal or greater than 10. A similar test data set may be used for the following C code for the **while** statement:

```
scanf("%d", &number);  
while (number < 10)  
    statement_1;  
statement_2;
```

During testing, we should also use special data and illegal data as test data. Programs should be tested in isolated modules or functions. Each module should be tested correctly before integrating into the main program. The whole program can then be tested. When we design test data for program testing, we need to select

test data that could make the program fail, so that errors can be rectified and the program be made more robust.

For example, consider a problem on computing the power of a positive integer number, x^y . After analyzing the problem, we have:

Purpose: Compute the power of a positive integer.

Required input: (1) the value for x ; and (2) the value for y .

Required output: The result of the computation.

Formula: The formula is x^y , where x is a positive integer number.

We can then design the algorithm for the problem, and implement the program. In program testing, we can design the test data set as shown in Table 2.1. The typical test data is first given. This occurs when both x and y are positive integer numbers. Then, all the special cases are considered. It occurs when x equals to zero or negative, or when y is zero or negative.

Table 2.1 Test data set.

Case	Inputs		Expected outputs
	x	y	
Normal case	2	4	16
Special cases	0	4	0
	2	0	1
	2	-4	0.0625
	-2	4	error

2.8 Documentation

As it is necessary to come back to the program for further error debugging, program maintenance or future program enhancement, documentation is an important step in the program development process. Proper documentation should include the following:

- the problem definition and specification;
- program inputs, outputs, constraints and mathematical equations;
- a flowchart or pseudocode for the algorithm;
- a source program listing;
- a sample test run of the program; and
- a user manual for end users.

The source program listing documents low-level implementation of the program. It is important to write programs that are simple, readable and easy to

understand. Thus, we need to have good programming style. Some of the recommendations are given as follows:

- Indentation, blank spaces and blank lines should be used to separate different conceptual sections of code.
- Meaningful names for variables and constants should be used. For example, in the statement

```
T = y*365 + m1*30 + m2*31 + d;
```

all the symbols used for variables are not very meaningful. The statement can be modified to reflect the real meaning of each variable by using meaningful symbols:

```
totaldays = noofYears*365 + noofMonths1*30 +  
noofMonth2*31 + noofDays;
```

- Programming style of one statement per line should be used.
- Program functions, braces and statements should be aligned properly to make the program readable.
- Lowercase letters including the underscore (_) should be used for variables and functions.
- Uppercase letters should be used for symbolic constants.
- Comments should be added into the program for the following situations: At the beginning of the program, comments should be given to state what the program will do, the author and date. Subsequently when the program is modified/extended, we should put comments to record the modifications, who made it, and the date of the modifications. If the purpose of a section of the program is not very clear at a glance, e.g. a nested loop or nested **if** statement, comments should be given to state the purpose of that section of program. We should also put comments at any place inside the program where it helps the reader to understand the program.

2.9 Exercises

1. List the six steps involved in program development process.
2. What is an algorithm? What are flowcharts and pseudocode?
3. What is top-down stepwise refinement in the context of program design?

4. A program is required to handle polynomial arithmetic. The operations are addition, subtraction and multiplication between two polynomials of degree 5. A polynomial is specified as (a0 a1 a2 a3 a4 a5) to represent $a_0 + a_1*x + a_2*x^2 + a_3*x^3 + a_4*x^4 + a_5*x^5$. The program will print a menu as follows:

```
(1) addition
(2) subtraction
(3) multiplication
(4) quit
Please type your operation =>
```

After the user indicates his choice of operation (1 to 3), the program will read two polynomials and produce the result to print. Then the whole process will repeat until the user types '4' to quit. Illustrate the top-down stepwise refinement design for the program to handle polynomial arithmetic.



Data Types, Constants and Variables

A program needs to work with *data*. A data object in a C program can be a variable or a constant. Variables can be used to store data whose values change during the execution of a program, while constants can be used to store data whose values remain fixed during the execution of a program. Both constants and variables must be of a certain type. The C language provides a variety of data types for storing data for the needs of different programs. The basic data types in C are integer, floating point and character.

This chapter covers the following topics:

- 3.1 Basic Data Types
 - 3.2 The Integer Type
 - 3.3 The Floating Point Type
 - 3.4 The Character Type
 - 3.5 Constants
 - 3.6 Variables and Declarations
-

3.1 Basic Data Types

A data type describes the size (in terms of number of bytes in memory) of an object and how it may be used. The three basic data types in C are **char**, **int** and **float**. Each type has its own computational properties and memory requirements. Moreover, the same data type may take different sizes on different types of machines. The type **int** takes two bytes on personal computers, but four bytes on

larger machines. In addition to the basic data types, C also provides several variations that change the meaning of the basic data types when applied to them. For example, the types **short** and **long** can be applied to integers.

3.2 The Integer Type

Any data object that is an integer is in this category. The type **int** is a signed integer which may be positive, zero or negative. Operations on integers are exact and faster than floating point operations. In C, there are six ways to store an integer object in a computer for C. Therefore, six integer types are available as shown in Table 3.1.

Table 3.1 Integer data types.

Type Name	Meaning
short	short integer
int	integer
long	long integer
unsigned	unsigned integer
unsigned short	unsigned short integer
unsigned long	unsigned long integer

The memory requirements for the integer data type are machine dependent. Table 3.2 shows the amount of memory used for objects of integer data types for personal computers (PC) and DEC (Digital Equipment Corporation) VAX machines. For instance, PCs use 2 bytes to store an **int**, which allows a range in values from -32768 to $+32767$.

Table 3.2 Memory usage of integer data types.

Data Type	Personal Computer	DEC VAX
short	16 bits	16 bits
int	16 bits	32 bits
long	32 bits	32 bits
unsigned	16 bits	32 bits

Table 3.3 shows the range of values allowed for each type depends on the number of bits used (assuming 2's complement representation). In principle, we may treat the type **long** as 32 bits, **short** as 16 bits, and **int** as either 16 bits or 32 bits, depending on the machine's natural word size.

The type **short** is a signed type, which may use less storage than **int**. This can be used to save memory storage if only small numbers are needed. The type **long** is a signed integer that can be used to store integers that require a larger

Table 3.3 Value ranges of 1, 2 and 4 bytes.

Bit	Number of distinct value	Range
8	256	−128, 127
16	65536	−32768, 32767
32	4294967296	−2147483648, 2147483647

range than type `int`. The type `unsigned` is an integer that can only have a positive value. This can be used in situations where the data objects will never be negative. Therefore, when choosing which data type to be used for an object, we should select the type whose range is just enough to cover all the possible values of the object.

The file `<limits.h>` contains the local definition about the upper and lower limits of the data types on the machine. Table 3.4 gives some of the constants defined in `<limits.h>`.

Table 3.4 Constant definitions in `<limits.h>`.

Constant	Value	Meaning
<code>INT_MIN</code>	−32768	minimum value of <code>int</code>
<code>INT_MAX</code>	32767	maximum value of <code>int</code>
<code>UNIT_MAX</code>	65535	maximum value of <code>unsigned int</code>
<code>LONG_MIN</code>	−2147483648	minimum value of <code>long</code>
<code>LONG_MAX</code>	2147483647	maximum value of <code>long</code>
<code>ULONG_MAX</code>	4294967295	maximum value of <code>unsigned long</code>
<code>SHRT_MIN</code>	−32768	minimum value of <code>short</code>
<code>SHRT_MAX</code>	32767	maximum value of <code>short</code>
<code>USHRT_MAX</code>	65535	maximum value of <code>unsigned short</code>

3.3 The Floating Point Type

Any data object, which is a real number, is in this category. Floating point number representation is similar to scientific notation that is especially useful in expressing very small or very large numbers. Exponential notation is used by most computers to represent floating point numbers. Table 3.5 gives a few examples of some numbers written in floating point number representation and exponential notation.

There are three ways to store a floating point object for C in the computer. Therefore, three C floating point types are available as shown in Table 3.6. They are `float`, `double` and `long double`.

Table 3.7 gives the amount of memory used for objects of these types that are machine dependent. The range of values allowed for each type depends on the number of bits used.

Table 3.5 Floating point number representation and exponential notation.

Number	Floating Point Number	Exponential Notation
123.321	1.23321×10^2	1.23321e2
0.0000123	1.23×10^{-5}	1.23e-5
123,000,000	1.23×10^8	1.23e8
123	1.23×10^2	1.23e2

Table 3.6 Memory used for floating point data types.

Type Name	Meaning
float	floating point
double	floating point with higher precision, i.e. with larger fractional part
long double	floating point with even higher precision

Table 3.7 The floating point data types.

Data Type	Personal Computer	VAX DEC
float	32 bits	32 bits
double	64 bits	64 bits
long double	not applicable	not applicable

The data type **float** is used to represent single-precision floating point. As shown in Table 3.7, 32 bits are used to store an object of type **float**. Eight bits are used for the exponent, and 24 bits are used to represent the nonexponential part. It typically has six or seven digits of precision and a range of 10^{-37} to 10^{+38} . The data type **double** is used to store double-precision floating point numbers. It normally uses twice as many bits of memory as **float**, typically 64 bits. ANSI C also allows for a third floating point type called **long double**. It aims to provide for even more precision than **double**.

The file `<float.h>` contains the local definition about the upper and lower limits of the data types on the machine. Table 3.8 gives the constant definitions in `<float.h>`.

The three numeric data types **int**, **float** and **double** are stored differently in the computer. The **int** data type requires the least memory space, the **double** data type requires the most memory space, and the **float** data type falls in between. Operations involving the **int** data type are always exact, while the **float** and **double** data types can be inexact. There is an infinite number of floating point numbers within each range covered by a type. Only some of these numbers can be represented exactly.

Table 3.8 Constant definitions in `<float.h>`.

Constant	Values	Meaning
DBL_MIN	1E-37	minimum positive double floating point number
DBL_MAX	1E+37	maximum double floating point number
DBL_DIG	10	decimal digits of precision
DBL_EPSILON	1E-9	machine precision for double
FLT_MIN	1E-37	minimum positive floating point number
FLT_MAX	1E+37	maximum floating point number
FLT_DIG	6	decimal digits of precision
FLT_EPSILON	1E-5	machine precision for float

3.4 The Character Type

Any data object, which is an English letter, an English punctuation mark, a decimal digit, or a symbol such as a space, is in this category. A character is actually stored internally as an integer. The conversion from characters to integer numbers is done commonly using the ASCII code. The ASCII (American Standard Code Information Interchange) code is the most commonly used character set. There are 128 distinct characters in the ASCII character set. Each character has a corresponding numeric ASCII code. For example, the character 'A' has the ASCII value of 65. The complete list of the ASCII character set is given in Appendix A.

There are two C character types: **char** and **unsigned char**. Both character types use 8 bits (1 byte) to store a character. So there are 256 possible values. The range of values is given in Table 3.9.

Table 3.9 Range of the character types.

Data Type	Range
char	-128, 127
unsigned char	0, 255

3.5 Constants

A constant is an object whose value is unchanged (or cannot be changed) throughout the program execution. There are four types of constants:

- integer constants, e.g. 100, -256;
- floating point constants, e.g. 2.4, -3.0;
- character constants, e.g. 'a', '+'; and
- string constants which are a string of characters, e.g. "have a good day".

Integer Constants

Integer constants may be specified in decimal, octal or hexadecimal notation. A decimal integer constant is written as a sequence of decimal digits (0-9), e.g. 1234. An octal integer constant is written as a sequence of octal digits (0-7) starting with a zero (0), e.g. 077. A hexadecimal integer constant is a sequence of hexadecimal digits starting with 0x or 0X, e.g. 0XFF. The hexadecimal digits comprise 0-9, and the letters A to F (or a to f), where the letters represent the values 10 to 15 respectively. A decimal, octal or hexadecimal integer constant can immediately be followed by the letter L (or letter l) to explicitly specify a **long** integer constant, e.g. 1234L and 0XFFL.

Floating Point Constants

Floating point constants may be written in two forms. The first is in the form of a sequence of decimal digits including the decimal point, e.g. 3.1234. The second is to use the exponential notation as discussed in the previous section, e.g. 3.1234e3. All floating point constants are always stored as **double**. If the floating point numbers are qualified by the suffix f or F, e.g. 1.234F, then the values are of type **float**. We may also qualify a floating point number with l or L to specify the value to be stored is of type **long double**.

Character Constants

Character constants can be given by quoting the numerical value of a character, e.g. 65 for the character **A** in the ASCII character set (see Appendix A), or by enclosing it with quotes, e.g. 'A'. Some useful non-printable control characters are referred to by the *escape sequence* which consists of the backslash symbol (\) followed by a single character. The two symbols together represent the single required character. This is a better alternative, in terms of memorization, than ASCII numbers. For example, '\n' represents the newline character instead of using the ASCII number 10. Table 3.10 gives some examples of escape sequence. Notice that escape sequences must be enclosed in single quotes.

Table 3.10 Examples of escape sequence.

Escape Sequence	Meaning	Escape Sequence	Meaning
'\a'	alarm bell	'\f'	form feed
'\t'	horizontal tab	'\"'	double quote
'\b'	back space	'\?'	question mark
'\''	single quote	'\\'	back slash
'\n'	newline	'\v'	vertical tab
'\r'	carriage return		

The escape sequence may also be used to represent octal and hexadecimal ASCII values of a character. The formats for octal and hexadecimal values are `'\000'` and `'\xhh'`. For example, `'\007'` and `'\x7'` are octal and hexadecimal values for the alarm bell. The character constant `'A'` (with ASCII encoding decimal 65, octal 101 or hexadecimal 41) may be represented as `'\0101'` or `'\x41'` for octal and hexadecimal values respectively.

A string constant is a sequence of characters enclosed in double quotation marks. It is different from a character constant. For example, `'a'` and `"a"` are different as `'a'` is a character while `"a"` is a string. Strings will be discussed in Chapter 10.

Defining Constants

There are three ways to define a constant. The first way is to define a constant by directly giving the value. An example is given in Program 3.1. In the `printf()` statement, `"The value for pi is %f.\n"` is a string constant, `3.14159` is a floating point constant.

Program 3.1 Defining a constant using a value directly.

```
#include <stdio.h>
main(void)
{
    printf("The value for pi is %f.\n", 3.14159);
    return 0;
}
```

Program output

```
The value for pi is 3.141590.
```

Another way to define a constant is to use a constant variable. This is done by using the `const` qualifier as follows:

```
const type varName=value;
```

where `type` can be `int`, `float`, `char`, etc. `varName` is the name of the constant variable. `value` is the constant value to be assigned to the constant variable. An example is given in Program 3.2. After the declaration of the statement

```
const float pi = 3.14159;
```


the value of 3.14159 will replace the constant variable **pi** wherever **pi** appears in the program.

Program 3.2 Defining a constant using a constant variable.

```
#include <stdio.h>
main(void)
{
    const float pi = 3.14159;
    printf("The value for pi is %f.\n", pi);
    return 0;
}
```

Program output

```
The value for pi is 3.141590.
```

The third way to define a constant is by using the preprocessor directive **#define**. The format of **#define** is

```
#define constantName value
```

where **constantName** is the name of the constant. Some examples are given as follows:

```
#define YES 'Y'
#define GREETINGS "How are you?"
#define ALARM '\a'
```

It is a C tradition that **constantName** is in upper case. It makes programs more readable. During compilation, the value of the constant will substitute the name of the constant whenever it appears in the program. By giving a symbolic name to a constant, it improves the readability of the program and makes the program easier to be modified. An example that uses **#define** to define a constant value is given in Program 3.3.

Program 3.3 Defining a constant using #define.

```
/* A program to compute the tax payable by a person */
#include <stdio.h>
#define TAXRATE 0.15
main(void)
{
    float salary, others, tax;
```

```
....  
tax = (salary + others) * TAXRATE;  
....  
  
return 0;  
}
```

C provides **#define** and **const** to define symbolic names for constants. We recommend using **#define** to name constants of simple data types, and **const** to define constants that depend on another constant. For example, in the declarations

```
#define TAXRATE 0.15  
const double monthrates=TAXRATE/12;
```

#define is used for defining the annual tax rate, while **const** is used to define the monthly tax rate which is one-twelfth of the annual tax rate.

3.6 Variables and Declarations

Programs work with data, and data is stored in memory. Variables are symbolic names that are used to store data in memory. The content in the memory location is the current value of the variable. We use the variable name in order to retrieve the value stored at the memory location. Variables are different from constants in that the value of a variable may change during program execution, while the value of a constant remains unchanged throughout the program execution. When a new value is assigned to the variable, the old value is replaced with the new value.

The name of a variable is made up of a sequence of alphanumeric characters and the underscore '_'. The first character must be a letter or an underscore. There is a system-dependent limit to the maximum number of characters that can be used. The ANSI C standard limit is 31.

C is also case sensitive. The variable names **count** and **COUNT** refer to two different variables. However, it is conventional to use lowercase letters to name variables. In addition, meaningful names make programs more readable. For example, **'tax'** will make a program easier to understand than **'t'**. A variable name cannot be any of the *keywords* in C. Keywords have special meanings to C compiler. Table 3.11 lists some of the keywords in C.

Each variable has a type. The basic C data types are integer, floating point and character. Variables are declared by *declaration statements*. A declaration can be done with or without initialization. A declaration statement without initialization has the format

```
type varName[, varName];
```

Table 3.11 Keywords in C.

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

where type can be **int**, **char**, **float**, **double**, etc. **varName** is the name of the variable. [...] may be repeated zero or more times but need to be separated by commas.

Program 3.4 shows an example of variable declaration. The type keywords **int**, **float** and **char** are used to declare variables. **tax** and **salary** are declared of the data type **float**, **numOfChildren** and **numOfParents** are declared of the data type **int**, and **maritalStatus** is declared of the type **char**. During program execution, memory storage of suitable size is assigned for each variable. A variable must be declared first before it is used. The memory location is used to store the current value of the variable. Variable names can then be used to retrieve the value stored in the memory location.

Program 3.4 Declaring variables.

```
main(void)
{
    float  tax, salary;
    int    numOfChildren, numOfParents;
    char   maritalStatus;
    ....
    return 0;
}
```

Initialization can also be done as part of a declaration. This means that a variable is given a starting value when it is declared. An example is given in Program 3.5. In this program, the variables **tax** and **salary** are declared without initial values. The other two variables **numOfChildren** and **numOfParents** are initialized to 2, and the variable **maritalStatus** is initialized to 'M' as shown in Figure 3.1.

Program 3.5 Declaring variables with initialization.

```
main(void)
{
    float  tax, salary;
```

```
int    numOfChildren = 2, numOfParents = 2;  
char   maritalStatus = 'M';  
....  
return 0;  
}
```

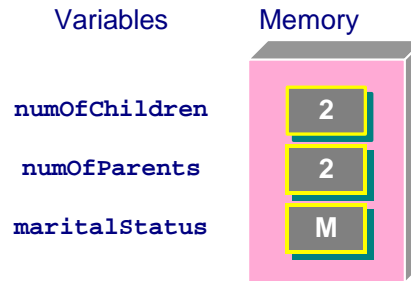


Figure 3.1 Initialized variables in memory.

The declaration statements for `numOfChildren`, `numOfParents` and `maritalStatus` declare and initialize the variables in one step. It is a very useful feature when developing complex programs.

The declaration statement

```
int numOfChildren, numOfParents = 2;
```

only initializes `numOfParents` to 2. `numOfChildren` is not initialized. To improve the readability of the program, it is better to declare initialized and uninitialized variables in separate declaration statements.

Also notice that a variable is not allowed to be declared twice. The declaration statements

```
float tax, salary;  
int tax;
```

will cause the compiler to issue an error message.

3.7 Exercises

1. Indicate whether each of the following variable names is valid. If invalid, state why.

(a) `3_persons`

(b) `three persons`

- (c) float
- (e) _wins
- (g) period.
- (d) special-rate
- (f) one_&_two
- (h) \$dollar

2. State the data type of each of the following:

- (a) '5'
- (c) 4.5
- (e) 0x45
- (g) 123456789L
- (i) -456
- (k) 0Xabc
- (b) 45
- (d) '\045'
- (f) '\n'
- (h) 4.567F
- (j) 0456
- (l) 0xaBcDeF12L

3. Find the errors in the following series of declarations:

```
main(void)
{
    int      variable, var;
    float    var = 1,25;
    integer  var2;
    int      long;
    short    $discount, var-2;
    ....
}
```

4. Describe the advantages of data type `int` over data types `float` and `double`.



Simple Input/Output

Most programs need to communicate with their environment. Input/output (I/O) is the way a program communicates with the user. For C, the I/O operations are carried out by the I/O functions in the standard I/O libraries. Input from the keyboard or output to the monitor screen is referred as standard input/output. In this chapter, we discuss four functions, which communicate with the user's terminal. The `printf()` and `scanf()` functions perform formatted I/O, while the `putchar()` and `getchar()` functions perform character I/O.

This chapter covers the following topics:

- 4.1 Formatted Input/Output
 - 4.2 The `printf()` Function
 - 4.3 The `scanf()` Function
 - 4.4 Character Input/Output
-

4.1 Formatted Input/Output

A function is a piece of code to perform a specific task. A library contains a group of functions, usually for related tasks. The standard I/O functions in the library `<stdio>` and the mathematical functions in the library `<math>` are two examples of library functions. To use the I/O functions in `<stdio>`, the preprocessor directive

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

must be included in a program. The two I/O functions that are most frequently used are the `printf()` and `scanf()` functions. `printf()` is used as an output

function, while `scanf()` is used as an input function.

4.2 The printf() Function

The `printf()` function is the most commonly used C library function. It allows us to control the format of the output. The `printf()` function can be used to print data of different data types in C language. An example of using the function `printf()` is given in Program 4.1.

Program 4.1 Using the printf() statement.

```
#include <stdio.h>
main(void)
{
    int num1 = 1, num2 = 2;

    printf("%d + %d = %d\n", num1, num2, num1+num2);
    return 0;
}
```

Program output

```
1 + 2 = 3
```

A `printf()` statement has the following format:

```
printf(control-string, argument-list);
```

The **control-string** which is enclosed in double quotation marks is a string constant. The string is printed on the screen. Two types of information are specified in the **control-string**. It comprises the characters that are to be printed and the conversion specifications (or format specifiers). In Program 4.1, `%d` is the *conversion specification*. It defines the ways the items are displayed on the screen. Conversion specifications can be placed anywhere within the **control-string**. Three conversion specifications with `%d` are illustrated in Program 4.1.

If the **control-string** is longer than one line, the continuation character `'\'` must be used:

```
printf("If a string is too long to fit on one line, \
      is used when it is written on two lines");
```

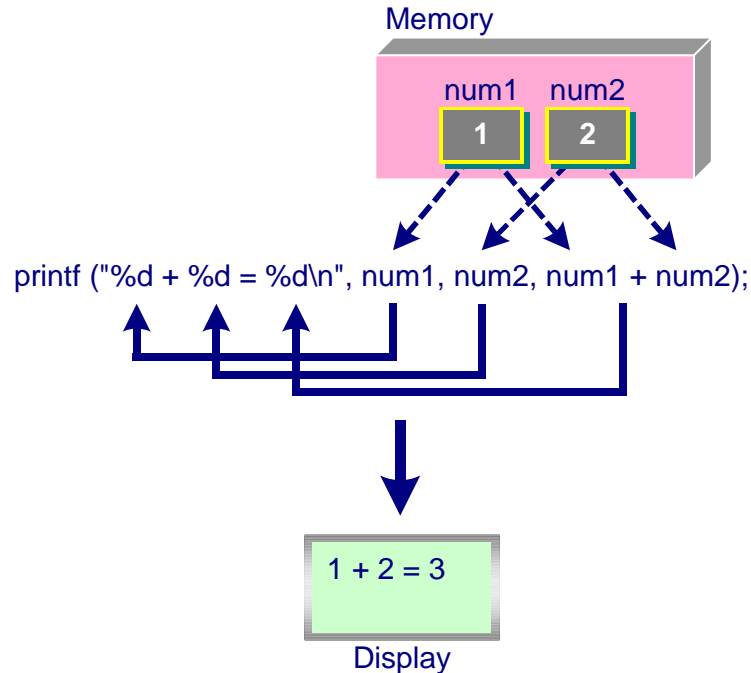


Figure 4.1 Conversion specification and argument-list.

The **argument-list** contains a list of items such as `item1`, `item2`, ..., etc. to be printed. They contain values to be substituted into places held by the conversion specifications in the **control-string**. This is illustrated in Figure 4.1.

An **item** can be a constant, a variable or an expression such as `num1+num2`. This is illustrated in Program 4.2. Notice that the number of items must be the same as the number of conversion specifications, and the type of the item must match with the **conversionSpecifier**.

Program 4.2 Another program using printf().

```
#include <stdio.h>
main(void)
{
    int num = 3;

    printf("The constant value = %d\n", 5);
    printf("The value of the variable = %d\n", num);
    printf("The result of the product 5*num = %d\n", 5*num);
    return 0;
}
```


Program output

```
The constant value = 5
The value of the variable = 3
The result of the product 5*num = 15
```

Conversion Specification

The format of a conversion specification is

```
%[flag][minimumFieldWidth][precision][sizeSpecification]
conversionSpecifier
```

where **%** and **conversionSpecifier** are compulsory. The others are optional. These options provide the additional control on how to print a value.

The **conversionSpecifier** specifies how the data is to be converted into displayable form. It consists of a conversion specifier that indicates the type of required conversion to be performed. Table 4.1 lists some of the conversion specifiers.

Table 4.1 Conversion specifiers in printf().

Conversion Specifier	Output
d	signed decimal integer
o	unsigned octal integer
u	unsigned decimal integer
x	unsigned hexadecimal integer, using lowercase letters, e.g. abcd
X	unsigned hexadecimal integer, using uppercase letters, e.g. ABCD
c	single character
f	signed floating point number, decimal notation
e	signed floating point number, e-notation
E	signed floating point number, E-notation
p	a pointer
g	use %f or %e format, whichever is shorter
G	use %f or %E format, whichever is shorter
%	print the % symbol

A **flag** can be any of the five values: **+**, **-**, **space**, **#** and **0**. Zero or more flags may be present. It controls the display of plus or minus sign of a number, the display of the decimal point in floating point numbers, and left or right justification. The **flags** are described in Table 4.2.

Table 4.2 Specification of flags.

Flag	Meaning
+	Signed values are printed with a plus sign if positive and a minus sign otherwise, e.g. "%+d".
-	Items are left justified, e.g. "%-d".
space	Signed values are printed with a leading space if positive and a minus sign otherwise, e.g. "% d".
#	For %o, it prints the value with an initial zero. For %x and %X, it prints the value with an initial 0x and 0X. For floating point numbers, it forces the printing of the decimal point. For %g and %G, it causes all trailing zeros to be printed. E.g. "%#o".
0	The field width is padded with zeros instead of spaces for numeric values, e.g. "%0d".

The **minimumFieldWidth** field gives the minimum field width to be used during printing. For example, "%10d" prints an integer with field width of 10. If the width of the item to be printed is less than the specified field width, then it is right-justified and padded with blanks or zeros. If the field width is not enough, a wider field will be used. For example, if the specification is "%2d", and the integer value is 123, which is 3 digits long, then the field width of 3 instead of 2 is automatically used.

The **precision** field follows the **minimumFieldWidth** and specifies the number of digits from the **minimumFieldWidth** after the decimal point. For %d, %u, %i, %o, %x and %X, the precision field specifies the number of digits to be printed. For %e, %E, and %f, the precision field specifies the number of digits to be printed to the right of the decimal. For %g and %G, the precision specifies the maximum number of significant digits. For %s, the precision determines the maximum number of characters to be printed. If only %.f is used, it means %.0f. For example, "%8.4f" prints a floating point number in a field 8 characters wide with 4 digits after the decimal point.

The **sizeSpecification** modifies the size of the data type specified by the **conversionSpecifier**. It consists of one of the following characters: h, l or L. Table 4.3 gives the **sizeSpecification** with the integer data type.

Examples of the printf() Statement

Program 4.3 prints three integers using conversion specification %d with **sizeSpecification** h and l. The program creates three variables i, j and k of data types **short**, **int** and **long** respectively. Initial values are assigned to

Table 4.3 sizeSpecification with integer data types.

Size Specification	Meaning
h	For %d , %i , %o , %x and %X , it indicates short integer, e.g. "%hx" . For %u , it specifies unsigned short integer, e.g. "%hd" .
l	For %d , %i , %o , %x and %X , it indicates long integer, e.g. "%ld" . For %u , it specifies unsigned long integer, e.g. "%ld" .
L	For %e , %E , %f , %g and %G , it indicates long double , e.g. "%Lf" .

these variables. The first `printf()` statement and the third `printf()` statement use the `%hd` and `%ld` specifiers to print the **short** and **long** integers. The second `printf()` statement uses just the conversion specification `%d` to print the integer.

Program 4.3 Printing integers.

```
#include <stdio.h>
main(void)
{
    short  i = 123;
    int    j = 123;
    long   k = 1234567890L;

    printf("short i = %hd\n", i);
    printf("integer j = %d\n", j);
    printf("long k = %ld\n", k);
    return 0;
}
```

Program output

```
short i = 123
integer j = 123
long k = 1234567890
```

Some more examples on printing integer values with different conversion specifications are given in Table 4.4. The integer value used in the examples is 125. The symbol `□` is used to represent a space in the output. In example (2), the output is right-justified and the positive sign is printed on the screen. In example (3), the output is left-justified. In example (4), the specified minimum field width

is less than the actual field width of the integer number, the number is displayed and no truncation is done.

Table 4.4 The printf() function for integer values.

	Conversion Specification	Flag	Minimum Field Width	Conversion Specifier	Output on Screen
(1)	%d	none	none	d	125
(2)	%+6d	+	6	d	□□+125
(3)	%-6d	-	6	d	125□□□
(4)	%1d	none	1	d	125

Program 4.4 prints four floating point numbers using the conversion specification %f with different options. The first two `printf()` statements print using %f. The default precision of 6 is used, i.e. six digits are printed after the decimal point. The third and fourth `printf()` statements are printed with different values of field width and precision. In the third `printf()` statement, the field width is limited to 4 with precision 1, i.e. only one digit is printed after the decimal point. Similarly in the fourth `printf()` statement, the field width is limited to 12 with precision 1. The floating point numbers to be printed are right-justified in the field.

Program 4.4 Printing floating point numbers.

```
#include <stdio.h>
main(void)
{
    float i = 12.3;
    double j = 123.4;

    printf("float i = %f\n", i);
    printf("double j = %f\n", j);
    /* by default, 6 digits are printed after the decimal
       point */

    printf("float i = %4.1f\n", i);
    printf("double j = %12.1f\n", j);
    return 0;
}
```

Program output

```
float i = 12.300000
double j = 123.400000
```

```
float i = 12.3
double j =          123.4
```

Program 4.5 prints three floating point numbers using the conversion specification `%f`. The first `printf()` statement prints a `'%'` character in front of the printed number, and one at the end of the printed number. The minimum field width specified is 20. The number to be printed is right-justified. In the second and third `printf()` statements, the minimum field width and precision are specified as 20 and 9. In addition, `"%-f"` format is used in the third `printf()` statement. The `"%-f"` format causes the number printed to be left-justified in the field.

Program 4.5 Printing floating point numbers with other specifications.

```
#include <stdio.h>
main(void)
{
    double k = 0.123456789;

    printf("double k = %%%20f%%\n", k);
    printf("double k = [%20.9f]\n", k);
    printf("double k = [%-20.9f]\n", k);
    return 0;
}
```

Program output

```
double k = %          0.123457%
double k = [          0.123456789]
double k = [0.123456789          ]
```

Further examples on printing floating point values with different conversion specifications are given in Table 4.5. The floating point value used in the examples is 12.345678. The symbol `□` represents a space. In example (1), the default precision of 6 is used. In example (2), the total number of digits is 11. The flag is positive, so the positive sign is printed. In example (4), the precision 4 is used, and the output is 12.3457 instead of 12.3456. In examples (5) and (6), the total number of digits is 12 and the precision is 3. In example (7), the precision is 2, and the field width is too short, therefore, the minimum field width is used for the output. In example (8), as the precision is not defined, the default precision of 6 is used for the output.

Table 4.5 The printf() function for floating point values.

	Conversion Specification	Flag	Minimum Field Width	Precision	Conversion Specifier	Output on Screen
(1)	%f	none	none	none	f	12.345678
(2)	%+11.5f	+	11	5	f	<input type="text"/> +12.34568
(3)	%-11.5f	-	11	5	f	12.34568 <input type="text"/>
(4)	%1.4f	none	1	4	f	12.3457
(5)	%+12.3e	+	12	3	e	<input type="text"/> +1.235e+001
(6)	%-12.3e	-	12	3	e	1.235e+001 <input type="text"/>
(7)	%4.2e	none	4	2	e	1.23e+001
(8)	%E	none	none	none	E	1.234568E+001

4.3 The scanf() Function

The **scanf()** function is an input function that can be used to read formatted data. The **scanf()** function reads input character strings from the input device (e.g. keyboard), converts the strings character-by-character to values according to the specified format and then stores the values into the variables. An example which shows the use of the **scanf()** statement is given in Program 4.6.

Program 4.6 Using the scanf() statement.

```
#include <stdio.h>
main(void)
{
    int num1,num2;

    printf("Enter 2 integers:\n");
    scanf("%d %d", &num1, &num2);
    printf("%d + %d = %d\n", num1, num2, num1+num2);
    return 0;
}
```

Program input and output

```
Enter two integers:
1 2
1 + 2 = 3
```

A **scanf()** statement has the following format:

```
scanf(control-string, argument-list);
```

control-string is a string constant containing conversion specifications. The **argument-list** contains the addresses of a list of input items that may be any variables matching the type given by the conversion specification. The input items cannot be a constant, or an expression such as **num1+num2**. The variable name has to be preceded by an address operator **&**. This is to tell **scanf()** the address of the variable so that **scanf()** can read the input value and store it in the variable. Commas are used to separate each input item in the argument-list. The **scanf()** statement reads data from the keyboard. It stops reading when it has read all the items as indicated by the **control-string** or the **EOF** (end-of-file) is encountered. On Unix systems, **EOF** is **^D** (control-D), while on DOS systems, **EOF** is **^Z** (control-Z).

scanf() uses whitespace characters (tabs, spaces and newlines) to determine how to separate the input into different fields to be stored. It ignores consecutive whitespace characters in between when matching up consecutive conversion specifications to different consecutive fields. Conversion specification that begins with a percent sign (%) reads characters from input, converts the characters into values of the specified format, and stores the values to the address memory locations of the specified variables. A conversion specification is of the form:

```
%[flag][maximumFieldWidth][precision][sizeSpecification]  
conversionSpecifier
```

where % and **conversionSpecifier** are compulsory. The others are optional. It is similar to the conversion specification of the **printf()** statement. Table 4.6 lists the main **conversionSpecifiers**.

Table 4.6 Conversion specifiers in **scanf()**.

Conversion Specifier	Expected Input
d	signed decimal integer
o	octal integer
u	unsigned decimal integer
x, X	hexadecimal integer
c	single character
e, E, f, g, G	floating point number in decimal or scientific format
p	a pointer input
s	character string
i	decimal, octal or hexadecimal integer

In the **scanf()** statement, the **flag** value of ***** means reading the next input

without assigning the value to the corresponding item, i.e. skipping instead of assigning the value to the next variable's memory location. The **maximumFieldWidth** in `scanf()` is the maximum number of characters to read rather than the minimum as in the `printf()` statement. The reading of input will also stop when the first whitespace is encountered. The **sizeSpecification** field consists of one of the characters **h**, **l** or **L**. It is mainly used to specify values of data type **short**, **long** or **double**. It overrides the default size for the type given in the **conversionSpecifier** field.

The `scanf()` function reads the input character one at a time. It skips over whitespace characters until it finds a nonwhitespace character. It then starts reading the characters until it encounters a whitespace character. However, `scanf()` will stop reading a particular input field under the following conditions:

1. If the **maximumFieldWidth** field is used, `scanf()` stops at the field end or at the first whitespace, whichever comes first.
2. When the next character cannot be converted into the expected format, e.g. a letter is read instead of a digit when the expected format is decimal. In this case, the character is placed back to the input stream. This means that the next input starts at the unread, nondigit character.
3. A matching nonwhitespace character is encountered.

Program 4.7 shows the use of different separators for input in the `scanf()` statement. The first `scanf()` statement uses the character `'/'` to separate user input. Other separation characters such as colon `':'`, hyphen `'-'` and comma `','` can also be used. Different **sizeSpecifications** `"%hd"` and `"%ld"` are used for values in **short** and **long** data types. The `scanf()` statement reads the characters until it matches the character `'/'`, converts them to a value of type **short** and assigns the value to the variable `num1`. The same process repeats for the next field, which converts the characters to an integer value and assigns the value to the variable `num2`. Then, it reads and converts the remaining data into a **long** integer and assigns the value to the variable `num3`. The operation of the `scanf()` function is illustrated in Figure 4.2.

Program 4.7 Using the `scanf()` statement with other specifications.

```
#include <stdio.h>
main(void)
{
    short    num1;
    int      num2;
    long     num3;
    float    real1, real4, real5, real6;
    double   real2;
```



```

long double real3;

printf("Enter a date: ");
/* to read in 3 decimal integers separated by / */
scanf("%hd/%d/%ld", &num1, &num2, &num3);
printf("The date is %2hd-%2d-%4ld\n\n", num1, num2,
       num3);

printf("Enter 3 real numbers:\n");
/* to read in 3 floating point numbers separated from
   each other by whitespaces, tabs or newlines. */
scanf("%f %lf %lf", &real1, &real2, &real3);
printf("They are %5.2f, %f, %8.4lf\n\n", real1, real2,
       real3);

printf("Enter 3 real numbers:\n");
scanf("%2f%2f%2f", &real4, &real5, &real6);
printf("They are %f, %f, %f\n", real4, real5, real6);
return 0;
}

```

Program input and output

```

Enter a date: 14/07/2002
The date is 14- 7-2002

Enter 3 real numbers:
74 7.88 666.56789
They are 74.00, 7.880000, 666.5679

Enter 3 real numbers:
74788C
They are 74.000000, 78.000000, 8.000000

```

The second **scanf()** statement uses a whitespace to separate user input. The **sizeSpecification** **"%lf"** is used to read a value in **double** data type. The third **scanf()** statement does not use any separators, the input data are entered one after the other. Notice that in the program output for the third **scanf()** statement, the nonwhitespace character 'C' is encountered instead of a digit. In this case, **scanf()** stops reading the input, and places the character 'C' back to the input stream. Thus, the variable **real6** in the third **scanf()** statement contains the value 8.000000.

The **scanf()** function can also return the number of items that it has successfully read. If no item is read, **scanf()** returns the value 0. If end-of-file is

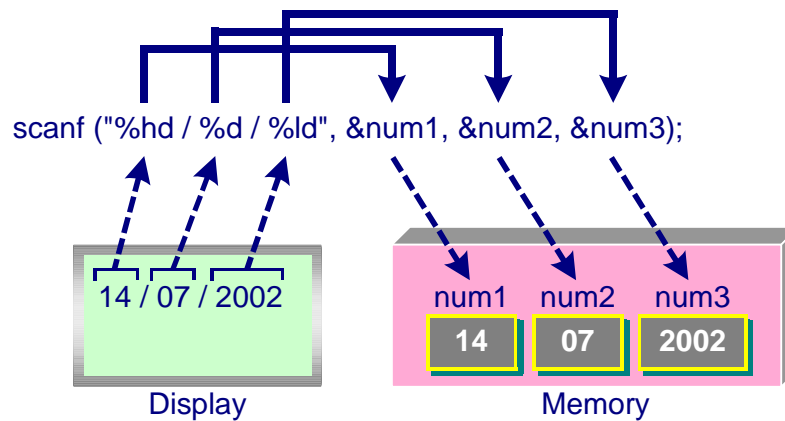


Figure 4.2 Operation of the scanf() function.

encountered, it returns **EOF**. We can modify the first **scanf()** statement in Program 4.7 to return the number of items read as

```
count = scanf("%hd/%d/%ld", &num1, &num2, &num3);
```

where **count** is a variable of data type **int**.

Consider the program shown in Program 4.8. The program implements input operation using the **scanf()** statement. The program first prompts the user to enter a number, it then reads in the number and asks the user whether the number is correct. The reply will be read in as a single character, i.e. **y** or **n** (yes or no) using **scanf()**, and will be stored in the variable **reply**. Then, the user's reply will be printed to the screen.

Program 4.8 A scanf() example that causes a reading problem.

```
#include <stdio.h>
main(void)
{
    char    reply;
    int     number;

    printf("Please enter a number: ");
    scanf("%d", &number);
    printf("The number read is %d\n", number);
    printf("Is the number correct (y/n)? ");
    scanf("%c", &reply);
    printf("Your reply is %c\n", reply);
    return 0;
}
```

```
}
```

Program input and output

```
Please enter a number: 13579
The number read is 13579
Is the number correct (y/n)? Your reply is
$
```

However, after the number has been entered and terminated with the **<Enter>** key, the terminating symbol (i.e. the newline '**\n**' character) remains in the input buffer. When the program prompts the user with the message "**Is the number correct (y/n)?**", it does not wait for the user input. Instead, the newline character in the input buffer is displayed as the reply by the user. This problem can be avoided by putting the **fflush(stdin)** function before **scanf()** as follows:

```
fflush(stdin);
scanf("%c", &reply);
```

This will flush or empty the input buffer that contains the newline character before asking for reply.

4.4 Character Input/Output

There are two functions in the **<stdio>** library to manage single character input and output: **putchar()** and **getchar()**. The function **putchar()** takes a single argument, and prints the character. The syntax of calling **putchar()** is

```
putchar(characterConstantOrVariable);
```

which is equivalent to

```
printf("%c", characterConstantOrVariable);
```

The difference is that **putchar()** is faster because **printf()** needs to process the control-string for formatting. Also, it needs to return either the integer value of the written character or **EOF** if an error occurs.

The **getchar()** function returns the next character from the input, or **EOF** if end-of-file is reached or if an error occurs. No arguments are required for **getchar()**. The syntax of calling **getchar()** is

```
ch = getchar();
```

where **ch** is a character variable. It is equivalent to

```
scanf("%c", &ch);
```

Program 4.9 shows an example to illustrate the use of **putchar()** and **getchar()**.

Program 4.9 Using **getchar()** and **putchar()**.

```
#include <stdio.h>
main(void)
{
    char ch, ch1, ch2;
    putchar('1');
    putchar(ch='a');
    putchar('\n');
    printf("%c%c\n", 49, ch);
    ch1 = getchar();
    ch2 = getchar();
    putchar(ch1);
    putchar(ch2);
    putchar('\n');
    return 0;
}
```

Program input and output

```
1a
1a
ab
ab
```

The **getchar()** function works with the input buffer to get user input from the keyboard. The input buffer is an array of memory locations used to store input data transferred from the user input. A *buffer position indicator* is used to keep track of the position where the data is read from the buffer. The **getchar()** function retrieves the next data item from the position indicated by the buffer position indicator and moves the buffer position indicator to the next character position. However, the **getchar()** function is only activated when the **<Enter>** key is pressed.

Therefore, when a key is pressed to enter a character, the input buffer receives and stores the input data until the **<Enter>** key is encountered. The **getchar()** function then retrieves the next unread character in the input buffer and advances

the buffer position indicator.

As illustrated in Program 4.9, when the user enters the data on the screen:

ab<Enter>

The input data, namely 'a', 'b' and '\n', will then be stored in the input buffer. The buffer position indicator points at the beginning of the buffer. After reading the two characters, 'a' and 'b', with the statements

```
ch1 = getchar();
ch2 = getchar();
```

the buffer position indicator moves two positions, and points to the buffer position that contains the newline '\n' character. Figure 4.3 illustrates the operation.

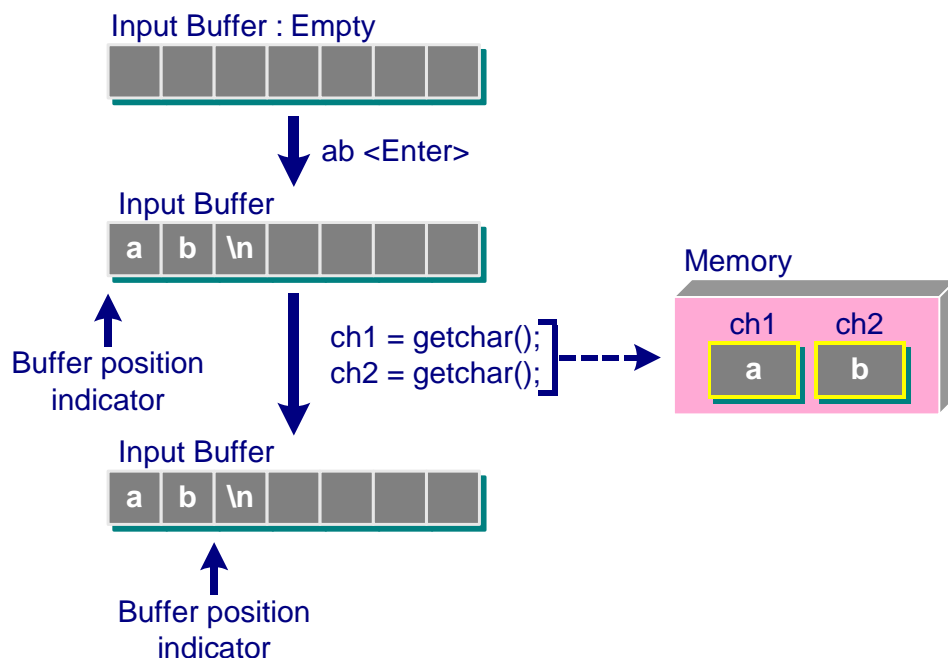


Figure 4.3 Operation of `getchar()` and the input buffer.

As illustrated in this example, the two `getchar()` functions execute and read in the character 'a' and 'b' only after the `<Enter>` key is pressed. However, the newline character (`\n`) still remains in the input buffer that needs to be taken care of before processing another input request. One way to deal with the extra newline character is to flush the input buffer by using the `fflush(stdin)` function to empty the input buffer before the next input operation.

4.5 Exercises

1. Assume the values for x is 2 and y is 3. What are the outputs of the following statements?

- (a) `printf("%d", x+x);`
- (b) `printf("x + x =");`
- (c) `printf("x + x = %d", x + x);`
- (d) `printf("%d = %d", x + y, y + x);`

2. What are the outputs of the following statements:

- (a) `printf("%c %x %d %o", 'X', 'X', 'X', 'X');`
- (b) `printf("abcd\r1234\b5\n");`
- (c) `printf("%c%c%c%c%c", 7, 007, 0x7, '\7', '\x7');`

3. Given the following declaration statements:

```
float f,g;
f = 1.2345;
g = 5.6789;
```

What are the outputs of the following statements:

- (a) `printf("f = %f g = %f", f, g);`
- (b) `printf("f = %6.2f g = %4.2f", f, g);`

4. Given the following declaration:

```
int day, month, year;
```

Use the `scanf()` statement to read the following input:

16/02/2002

and save them into the variables `day`, `month` and `year` respectively.



Operators, Expressions And Assignments

A C program contains a sequence of statements. A statement is a command to the computer. A statement can be either a simple statement or a compound statement. Assignment statements involve variables and expressions. Expressions involve constants, variables and operators. In this chapter, we present the way to write C programs that work on constants and variables. C statements can perform arithmetic operations such as addition, subtraction, multiplication and division. C provides a number of operators to perform these arithmetic operations. This chapter will discuss the arithmetic operators, increment and decrement operators, assignment operator and bitwise operators.

This chapter covers the following topics:

- 5.1 Operators
 - 5.2 Fundamental Arithmetic Operators
 - 5.3 Increment/decrement Operators
 - 5.4 Precedence of Operators
 - 5.5 Expressions
 - 5.6 Assignment Statements
 - 5.7 Data Type Conversion
 - 5.8 Mathematical Functions
 - 5.9 Bit Manipulation
-

5.1 Operators

An operator is a symbol that causes some operations to be performed on one or more variables (or data values). Operators in C can be classified into fundamental arithmetic operators, increment/decrement operators, relational operators, bitwise operators and conditional operators. In this chapter, we discuss the fundamental arithmetic operators, increment/decrement operators and bitwise operators. Relational operators and conditional operators will be covered in the next chapter. Operators can also be classified into unary operators, binary operators and ternary operators based on the number of operands they require.

5.2 Fundamental Arithmetic Operators

There are two types of fundamental arithmetic operators: unary operators and binary operators. Unary operators are positive (+), negative (-) (e.g. `+31`, `-5`). They can be used to change the sign of a value. Binary operators are addition (+), subtraction (-), multiplication (*), division (/) and modulus (%). Table 5.1 lists the common arithmetic operators.

Table 5.1 Arithmetic operators.

Operator	Meaning	Example
+	addition	<code>int z = 17+4; (z is 21)</code>
-	subtraction	<code>int z = 17-4; (z is 13)</code>
*	multiplication	<code>int z = 17*4; (z is 68)</code>
/	division	<code>int z = 17/4; (z is 4)</code>
%	modulus	<code>int z = 17%4; (z is 1)</code>
-	unary minus	<code>int z = -17; (z is -17)</code>

The division of floating point numbers (or one floating point number with an integer) returns a floating point number. However, the division operator (/) for integers returns integer results. The modulus operator (%) returns the remainder of an integer division. The modulus operator works only on data types `int` and `char`. The following examples give some of the arithmetic operations:

```

15.0/4 = 3.75
15/4.0 = 3.75
15/4 = 3
18/4 = 4
15%4 = 3
18%4 = 2
-18%4 = 2

```


It is also possible to perform arithmetic operations on character variables:

```
char chr1, chr2;
chr1 = 'A';          /* assigns 'A' to the variable chr1 */
chr2 = chr1 + 32;    /* adds 32 to the variable chr1 */
```

The second statement essentially converts the uppercase letter into the corresponding lowercase character. This is done by adding 32 to the ASCII code for the corresponding uppercase character. The result of the assignment statement is the character constant 'a' to be assigned to the variable **chr2**. This is a very useful technique in converting an uppercase letter to its lowercase counterpart, or vice versa.

5.3 Increment/decrement Operators

The increment operator (**++**) increases a variable by 1. It can be used in two modes: *prefix* and *postfix*. The format of the prefix mode is

```
++varName;
```

where **varName** is incremented by 1 and the value of the expression is the updated value of **varName**. The format of the postfix mode is

```
varName++;
```

where the value of the expression is the current value of **varName** and then **varName** is incremented by 1. Notice that one important difference between the prefix and postfix modes is on when the operation is performed. In the prefix mode, the variable is incremented before any operation with it, while in the postfix mode, the variable is incremented after any operation with it.

The decrement operator (**--**) works in a similar way as the **++** operator, except that the variable is decremented by 1. Table 5.2 summarizes the increment and decrement operators.

Table 5.2 Increment and decrement operators.

Operator	Meaning
++varName	increment varName before any operation with it (prefix mode)
varName++	increment varName after any operation with it (postfix mode)
--varName	decrement varName before any operation with it (prefix mode)
varName--	decrement varName after any operation with it (postfix mode)

Program 5.1 shows some examples on the use of increment/decrement

operators. The initial values of the two variables `num1` and `num2` are 5. The first statement prints the value of 5 for `num1`. The second statement also prints 5. As this statement uses the postfix mode, the value of `num1` is incremented by 1 after the printing has taken place. The third statement then prints the value of 6 for `num1`. The fourth statement increments the value of `num1` to 7, and the `printf()` statement operates on the new value of `num1` and thus, it prints the value of 7. The fifth statement also prints the value of 7 for `num1`. Similar operations are taken place for the statements using decrement operators on the variable `num2` in Program 5.1.

Program 5.1 Increment and decrement operators.

```
#include <stdio.h>
main(void)
{
    int num1 = 5, num2 = 5;

    printf("num1 = %d\n", num1);           /* first statement */
    printf("num1++ = %d\n", num1++);       /* second statement */
    printf("num1 = %d\n", num1);           /* third statement */
    printf("++num1 = %d\n", ++num1);        /* fourth statement */
    printf("num1 = %d\n\n", num1);          /* fifth statement */

    printf("num2 = %d\n", num2);
    printf("num2-- = %d\n", num2--);
    printf("num2 = %d\n", num2);
    printf("--num2 = %d\n", --num2);
    printf("num2 = %d\n", num2);

    return 0;
}
```

Program output

```
num1 = 5
num1++ = 5
num1 = 6
++num1 = 7
num1 = 7

num2 = 5
num2-- = 5
num2 = 4
--num2 = 3
num2 = 3
```

5.4 Precedence of Operators

Precedence of operators decides the order of evaluation for arithmetic expressions containing several operators. Each operator is assigned with precedence. The list of operators with decreasing priority is given in Table 5.3. The order of evaluation for operators with the same priority is generally from left to right (they associate from left to right). As listed in Table 5.3, the parentheses, the unary operator, as well as the increment and decrement operators have the highest precedence. They are followed by the multiplication, division and modulus operators. Then, the next in the order of precedence are addition and subtraction operators. The assignment operator has the lowest precedence.

Table 5.3 Operator precedence.

Operator	Meaning	Associativity
()	parentheses	left to right
+, -	unary	left to right
++, --	increment, decrement	right to left
*, /, %	multiplication, division, modulus	left to right
+, -	binary addition, subtraction	left to right
=	assignment	right to left

Consider the following statement:

```
a = ((x + y) * (x - y)) / z;
```

First, the parentheses have the highest precedence. The parentheses in **(x + y)** and **(x - y)** are evaluated first. The evaluation will be done in the direction from left to right. Next, the parentheses of the two results with the multiplication operator will be evaluated. Finally, the division will be evaluated. As illustrated, it is necessary to have parentheses in some cases in order to evaluate the expression correctly.

5.5 Expressions

An expression may be a constant (e.g. **2**, **'h'**, **-6.7**), a variable (e.g. **tax**, **salary**), or a combination of operators and operands (e.g. **a+b/c**). An expression can also involve incrementing or decrementing other variables. The following are a few examples on the evaluation of expressions, which are executed sequentially.

```
if num is 5, then
    num++          => the value of the expression is 5 (num becomes 6)
```

```
3 * num--    => the value of the expression is 18 (num becomes 5)
--num * 2    => the value of the expression is 8 (num becomes 4)
2 * ++num    => the value of the expression is 10 (num becomes 5)
```

An expression may also contain a function or involve assignments:

```
5 + sqrt(y)
num + (h = 5)
```

However, we should avoid using the increment/decrement operators on a variable, which appears more than once in an expression. The following should be avoided:

```
num++ * num++
num-- * num
```

5.6 Assignment Statements

An assignment statement is a statement to assign a value to a variable. It is the basic building block of a program. The value of a variable may be changed by the assignment statement in the following format:

```
varName = expression;
```

where **varName** is a variable name, and the **expression** (on the right-hand side) provides a value that is assigned to the variable **varName** on the left-hand side. Notice that the left-hand side must be a variable name and cannot be a constant or expression with operators. The expression on the right-hand side can be a constant, a variable or an arithmetic expression. The symbol **=** in the assignment statement is called the assignment operator. It does not mean equal to, instead it means *assigned to*.

An example on assignment statement is given in Program 5.2. When the program starts executing the first statement, the value 3 is stored in the memory location indicated by the variable **numOfChildren**. When the second, third and fourth statements are executed, the values 8000.00, 0.15 and 5678.90 are assigned to the memory locations of the variables **salary**, **taxRate** and **account**. The statement

```
account = account - salary * taxRate;
```

is written in such a way that the variable **account** appears on both sides of the assignment operator. The statement means that the value stored at the memory location **account** is retrieved, and the value is then used in the evaluation of the

expression **account-salary*taxRate**, the result is then assigned back to the memory location of the variable **account**. The previous value of **account** is lost after the assignment operation. In this case, the value 5678.90 is lost, and the value of the expression which equals to 4478.90 is now stored in the variable **account**.

Program 5.2 Using assignment statements.

```
main(void)
{
    int numOfChildren;
    float account, salary, taxRate;

    numOfChildren = 3;
    salary = 8000.00;
    taxRate = 0.15;
    account = 5678.90;
    ....
    account = account - salary * taxRate;
    ....
    return 0;
}
```

The lvalue and the rvalue

The assignment operator has two operands. The operand on the left-hand side must be a variable or an expression that names the memory location. The expression on the right-hand side can be a variable, a constant or an expression that gives a value to be assigned to the left-hand side. When a variable, e.g. **num**, is declared, the compiler allocates a memory location for storing the value of that variable. The *lvalue* of the variable refers to the address of the memory location to receive the value. Consider the following assignment statement:

```
num = 3;
```

the lvalue of **num** is used to store the value 3. The *rvalue* of the variable refers to the value or content of the memory location, i.e. the actual value of the variable. In the following assignment statement:

```
count = num;
```

the rvalue of **num** is used and assigned to the variable **count**.

The statements

```
3 = num;
```

```
num + count = 3;
```

are illegal as the left-hand side is not memory location. In the first statement, the left-hand side is a constant and in the second statement, the left-hand side is an arithmetic expression, which does not represent a memory location.

Arithmetic Assignment Operators

In addition to the assignment operators, C also has arithmetic assignment operators, which combine an arithmetic operator with the assignment operator. Arithmetic assignment operator statement has the following format:

```
varName op= expression;
```

where the arithmetic assignment operators (**op=**) are **+=**, **-=**, ***=**, **/=** and **%=**. This is equivalent to

```
varName = varName op expression;
```

The arithmetic assignment operator first performs the arithmetic operation specified by the arithmetic operator, and assigns the resulting value to the variable. Table 5.4 gives some examples of arithmetic assignment operator statements and its corresponding meaning.

Table 5.4 Arithmetic assignment operators.

Arithmetic Assignment Operator	Meaning
<code>varName += expression;</code>	<code>varName = varName + expression;</code>
<code>varName -= expression;</code>	<code>varName = varName - expression;</code>
<code>varName *= expression;</code>	<code>varName = varName * expression;</code>
<code>varName /= expression;</code>	<code>varName = varName / expression;</code>
<code>varName %= expression;</code>	<code>varName = varName % expression;</code>

Program 5.3 shows an example on the use of the arithmetic assignment operator **+=**. The statement

```
account += income;
```

is equivalent to

```
account = account + income;
```

The assignment statements involving increment/decrement operators are listed in Table 5.5. Program 5.4 gives an example on the use of assignment statements

Program 5.3 Using arithmetic assignment operator +=.

```
#include <stdio.h>
main(void)
{
    float account = 2000.00, income = 1000.00;

    printf("account = %8.2f, income = %8.2f\n", account,
        income);
    account += income;
    printf("account = %8.2f, income = %8.2f\n", account,
        income);
    return 0;
}
```

Program output

```
account = 2000.00, income = 1000.00
account = 3000.00, income = 1000.00
```

using increment and decrement operators. When we use the increment and decrement operators in assignment statements, we need to be careful whether we are using the prefix or postfix mode. In the prefix mode, i.e. when the ++/-- operator is placed before the variable **counter**, the variable is incremented/decremented first, the new value is then used to evaluate the expression and assign the result to another variable **num**. In the postfix mode, i.e. when the ++/-- operator is placed after the variable **counter**, the current value of the variable **counter** is used to evaluate the expression and assign the result to another variable **num**, the variable **counter** is then incremented/decremented.

Table 5.5 Assignment with decrement/decrement operators.

Assignment Operators	Meaning
VarName1 = varName2++;	varName1 = varName2; varName2 = varName2 + 1;
VarName1 = ++varName2;	varName2 = varName2 + 1; varName1 = varName2;
VarName1 = varName2--;	varName1 = varName2; varName2 = varName2 - 1;
VarName1 = --varName2;	varName2 = varName2 - 1; varName1 = varName2;

Program 5.4 Using assignments with increment/decrement operators.

```
#include <stdio.h>
main(void)
{
    int num = 10, counter = 20;

    printf("num = %d, counter = %d\n", num, counter);
    num++;
    ++counter;
    printf("num = %d, counter = %d\n", num, counter);
    num = 0, counter = 10;
    printf("num = %d, counter = %d\n", num, counter);
    num = counter--;
    /* decrement assignment operator statement (postfix) */
    printf("num = %d, counter = %d\n", num, counter);
    num = --counter;
    /* decrement assignment operator statement (prefix) */
    printf("num = %d, counter = %d\n", num, counter);
    return 0;
}
```

Program output

```
num = 10, counter = 20
num = 11, counter = 21
num = 0, counter = 10
num = 10, counter = 9
num = 8, counter = 8
```

For example, if we define two variables **m**, **k** as follows:

```
int m, k=2;
```

the statement

```
m = k++;
```

which is the same as

```
m = k;
k = k + 1;
```

will give **m** the value of 2 and **k** the value of 3 after executing the statement. However, the statement


```
m = ++k;
```

which is the same as

```
k = k + 1;  
m = k;
```

will give **m** the value of 3 and **k** the value of 3 after executing the statement.

Multiple assignments are also possible in one statement:

```
var1 = var2 = ... = varN = expression;
```

This is equivalent to

```
varN = expression;  
....  
var2 = expression;  
var1 = expression;
```

For example, in the following assignment statement:

```
num = 5 * (temp = 3) - 1;
```

the variable **temp** is assigned with an integer value 3. The expression (5 * 3 - 1) is then evaluated and assigned to the variable **num**.

5.7 Data Type Conversion

Data type conversion is the conversion of one data type into another type. It is needed when more than one type of data objects appear in an expression/assignment. For example, the statement

```
a = 5 + 2.68;
```

adds two numbers with different data types, i.e. *integer* and *floating point*. However, the addition operation can only be done if these two numbers are of the same type.

Type conversion will take place when two operands of an expression are of different data types. Three kinds of conversion can be performed: (1) explicit conversion; (2) arithmetic conversion; and (3) assignment conversion.

In *explicit conversion*, it uses the type *cast* operators, (**type**), e.g. (**int**), (**float**), (**double**), etc. to force the compiler to convert the value of the operand into the type specified by the operator. For example, the statements

```
int num;  
float result;  
result = float(num);
```

explicitly convert the value of **num** into data type **float** and assign the value to the variable **result** of data type **float**.

In *arithmetic conversion*, it performs automatic conversion in any operation that involves operands of two different types. It converts the operands to the type of the higher ranking of the two. This process is also called *promotion*. The ranking of types is based on the amount of memory storage the value needs. The ranking from high to low is given as follows:

```
long double > double > float > long > int > char
```

Consider the following statements:

```
float ans1, ans2;  
ans1 = 1.23 + 5/4;    /* first statement */  
ans2 = 1.23 + 5.0/4; /* second statement */
```

In the first statement, the expression **5/4** will be evaluated using integer division. It gives the result of 1. This result is then converted to 1.0 and added to the floating point constant 1.23 to give the final result of 2.23. This value is then assigned to the variable **ans1**. In the second statement, the expression **5.0/4** will be evaluated using floating point arithmetic. The value 4 in the expression will be converted to 4.0 before evaluating the expression. The result of 1.25 is then added to 1.23 to give the final result of 2.48. The result is then assigned to the variable **ans2**.

In *assignment conversion*, it converts the type of the result of computing the expression to that of the type of the left-hand side if they are different. If the variable on the left-hand side of the assignment statement has a higher rank or same rank as the expression, then there is no loss of precision. Otherwise, there can be a loss of accuracy. For example, if an expression has a floating point result, which is assigned to an integer variable, the fractional part of the result will be lost. This is because the lower-ranking type may not have enough memory storage to store the value of higher-ranking type. For example, the statements

```
int i;  
float x=2.5, y=5.3;  
i = x + y;
```

will give **i** a value of 7.

Program 5.5 shows an example program on data type conversion. In the first

assignment statement, it performs explicit conversion using type *cast* operator (`int`). It forces the compiler to convert the value into the specified data type `int` first before the addition operation. Both 2.5 and 3.7 are converted to integers before addition is performed. The value 2.5 is converted to 2 and the value 3.7 is converted to 3. Integer arithmetic is then performed and the result is 5. The result is then assigned to the variable `num`. In the second assignment statement, assignment conversion is performed. The two floating point values are first added, then the sum is converted into the integer variable `num`. In the last `printf()` statement, the addition operation of 2+3.7 first converts 2 to a higher ranking type, i.e. floating point type. The value of 2.0 is obtained, and then this value is added to 3.7 to get 5.7. The floating point result of 5.7 is then printed. In general, the programmer needs to be fully aware of implicit conversion, as such conversion may cause unexpected results.

Program 5.5 Data type conversion.

```
#include <stdio.h>
main(void)
{
    int num;
    float num2;

    num = (int)2.5 + (int)3.7;    /* explicit conversion */
    /* convert 2.5 to 2 and 3.7 to 3, then perform addition */
    printf("num = %d\n", num);

    num = 2.5 + 3.7;             /* assignment conversion */
    /* add 2.5 and 3.7 to get 6.2, then convert it to 6 */
    printf("num = %d\n", num);

    num2 = 2 + 3.7;              /* arithmetic conversion */
    /* convert 2 to 2.0, then perform addition */
    printf("num2 = %f\n", num2);
    return 0;
}
```

Program output

```
num = 5
num = 6
num = 5.700000
```

5.8 Mathematical Functions

All C compilers provide a set of mathematical functions in the standard library. Some of the common mathematical functions include square root `sqrt()`, power `pow()`, and absolute values `abs()` and `fabs()`. The `sqrt()` function computes the square root of a value of type `double`. It returns the result of type `double`. The statement

```
y = sqrt(x);
```

computes the square root of the value stored in the variable `x`, returns and stores the result in the variable `y`. The functions `abs()` and `fabs()` give the absolute values of an integer and a floating point number respectively. The statements

```
y = abs(-5) + 3;
y = fabs(-5.5) + 3.5;
```

call the functions `abs()` and `fabs()`, and return the values 5 and 5.5 respectively. The `pow()` function will take in two arguments, `x` and `y`, and returns x^y . Apart from the above functions, trigonometric functions such as `sin()`, `cos()` and `tan()` that compute the sine, cosine and tangent of an angle are also provided.

However, in order to use any of the mathematical functions, we need to place the preprocessor directive

```
#include <math.h>
```

at the beginning of the program. Table 5.6 gives some of the mathematical library functions that are available in the `<math>` library.

Table 5.6 C mathematical library functions.

Function	Argument Type	Description	Result Type
<code>ceil(x)</code>	<code>double</code>	Return the smallest <code>double</code> larger than or equal to <code>x</code> that can be represented as an <code>int</code> .	<code>double</code>
<code>floor(x)</code>	<code>double</code>	Return the largest <code>double</code> smaller than or equal to <code>x</code> that can be represented as an <code>int</code> .	<code>double</code>
<code>abs(x)</code>	<code>int</code>	Return the absolute value of <code>x</code> , where <code>x</code> is an <code>int</code> .	<code>int</code>

fabs(x)	double	Return the absolute value of x , where x is a floating point number.	double
sqrt(x)	double	Return the square root of x , where x ≥ 0 .	double
pow(x,y)	double x, double y	Return x to the y power, x^y .	double
cos(x)	double	Return the cosine of x , where x is in radians.	double
sin(x)	double	Return the sine of x , where x is in radians.	double
tan(x)	double	Return the tangent of x , where x is in radians.	double
exp(x)	double	Return the exponential of x with the base e, where e is 2.718282.	double
log(x)	double	Return the natural logarithm of x .	double
log10(x)	double	Return the base 10 logarithm of x .	double

5.9 Bit Manipulation

C allows programmers to interact with the hardware system through bitwise operators and expressions. When performing bit manipulation, data objects are viewed as binary strings, i.e. 0's and 1's. C supports six operators to manipulate bit values of integral types. The operators are listed in Table 5.7. The operators work only with integer types such as **char** and **int**. Thus, these operators may not be applied to data types **float** and **double**. The operators listed in Table 5.7 have the decreasing order of priority with *bitwise or* and *bitwise xor*, and *left shift* and *right shift*, having the same priority. The associativity for all the operators is from left to right. We restrict our discussion to a machine with a 16-bit for storing an **int**, using two's complement representation.

Table 5.7 Bitwise operators.

Category	Operator	Symbol	Description
Bitwise operators	bitwise complement	~	This operator converts a 1 in each bit position to a 0 and a 0 in each bit position to 1.
	bitwise AND	&	This operator sets a 1 in each bit position if the corresponding bits in the two operands are both 1.

Shift operators	bitwise OR		This operator sets a 1 in each bit position if at least one of the corresponding bits in the two operands is 1.
	bitwise XOR	^	This operator sets a 1 in each bit position if exactly one of the corresponding bits in the two operands is 1.
	left shift	<<	This operator shifts the bits of the first operand to the left by the number of bits specified by the second operand.
	right shift	>>	This operator shifts the bits of the first operand to the right by the number of bits specified by the second operand.

The *bitwise complement operator* (\sim), also known as the one's complement operator, converts a 1 in each bit position to a 0 and a 0 in each bit position to 1. Every bit in the binary representation of the result is the inverse of the operand. For example, if the `int` variable `x` has the value of 8, the binary representation for this value is 0000000000001000, the bitwise complement $\sim x$ is 1111111111111011. The decimal value of $\sim x$ is -9 (assuming 2's complement). The *bitwise operators and* ($\&$), *or* ($|$) and *exclusive or* (\wedge) are binary operators. Table 5.8 gives the definitions of these three operators. Table 5.9 shows the example expressions, the binary representation of the result and the corresponding decimal representation.

Table 5.8 Operations of the bitwise operators.

<code>x</code>	<code>y</code>	<code>x & y</code>	<code>x y</code>	<code>x ^ y</code>
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Table 5.9 Example expressions.

Expression	Binary representation	Decimal representation
<code>a</code>	0000000000001100	12
<code>b</code>	111111111011101	-35
<code>a & b</code>	0000000000001100	12
<code>a b</code>	111111111011101	-35

$a \wedge b$

111111111010001

-47

The shift operators move bits right or left. The first (or left) operand to a shift operator holds the bits to be shifted, and the second (or right) operand tells how far to shift the bits. Before shifting the bits, the first operand is converted to an `int`. The *left shift operator* (`<<`) shifts the bits of the first operand to the left by the number of bits specified by the second operand. The *right shift operator* (`>>`) shifts the bits of the first operand to the right by the number of bits specified by the second operand. The bits shifted in from the left are machine-dependent. If the first operand is unsigned, the bits are shifted from the left with 0 bit. If the first operand is a signed operand, then whether 0's or 1's are shifted in depends on system implementation. For left shift operation, the bits shifted from the right with 0 bit. Assume that the variable `x` has the value of 12. Table 5.10 shows a number of expressions, the binary representation of the result and the corresponding decimal representation. The bitwise shift operators are very useful in data compression for saving memory storage space.

Table 5.10 Shift operators.

Expression	Binary representation	Decimal representation
<code>x</code>	0000000000001100	12
<code>x << 4</code>	0000000011000000	192
<code>x >> 2</code>	0000000000000011	3

Program 5.6 illustrates the bitwise operations. A menu is provided for users to select the bitwise operations to be performed. Once the choice of an operation is selected, the program performs the corresponding operation, and the result will be printed on the display.

Program 5.6 Bitwise operations.

```
#include <stdio.h>
main(void)
{
    int x, y;
    int choice;
    int shift_left, shift_right; /* the number of places to
        shift */

    printf("Bit operations: \n");
    printf("1. Bitwise complement \n");
    printf("2. Bitwise and \n");
    printf("3. Bitwise or \n");
    printf("4. Bitwise xor \n");
```

```

printf("5. Left shift \n");
printf("6. Right shift \n");
printf("Enter your choice: ");
scanf("%d", &choice);

if ((choice == 1) || (choice == 5) || (choice == 6)) {
    printf("Enter a hex number => ");
    scanf("%X", &x);
}
else {
    printf("Enter 2 hex numbers => ");
    scanf("%X %X", &x, &y);
}

switch (choice)
{
    case 1: printf("~x = %X\n", ~x);
            break;
    case 2: printf("%X & %X = %X\n ", x, y, x & y);
            break;
    case 3: printf("%X | %X = %X\n ", x, y, x | y);
            break;
    case 4: printf("%X ^ %X = %X\n ", x, y, x ^ y);
            break;
    case 5: printf("Enter the no. of bits to be left-
                shifted: ");
            scanf("%X", &shift_left);
            printf("%X << %d = %X\n", x, shift_left,
                x<<shift_left);
            break;
    case 6: printf("Enter the no. of bits to be right-
                shifted: ");
            scanf("%X", &shift_right);
            printf("%X >> %d=%X\n", x, shift_right,
                x>>shift_right);
            break;
}
return 0;
}

```

Program input and output

Bit operations:
 1. Bitwise complement
 2. Bitwise and
 3. Bitwise or


```

4. Bitwise xor
5. Left shift
6. Right shift
Enter your choice: 1
Input a hex number => AB
~x = FFFFFFF54

Bit operations:
1. Bitwise complement
2. Bitwise and
3. Bitwise or
4. Bitwise xor
5. Left shift
6. Right shift
Enter your choice: 4
Input 2 hex numbers => AB 12
AB ^ 12 = B9

Bit operations:
1. Bitwise complement
2. Bitwise and
3. Bitwise or
4. Bitwise xor
5. Left shift
6. Right shift
Enter your choice: 5
Input a hex number => 12
Enter the no. of bits to be left-shifted: 3
12 << 3 = 90

```

5.10 Exercises

1. Write the following algebraic expressions in simple C arithmetic expressions:

(a)
$$\frac{a + b}{x + (c - d)}$$

(b)
$$\frac{a + b}{x^2(c + d)^2}$$

2. Given the following declarations:

```

int i;
i = 5;

```

What are the outputs of the following statements. The statements are executed sequentially.

- (a) `printf("i = %d", ++i);`
- (b) `printf("i = %d", --i);`
- (c) `printf("i = %d", i++);`
- (d) `printf("i = %d", i--);`

3. Given the following declarations:

```
int i=10, j=3;
float a=1.2;
```

Compute the values of the following expressions:

- | | |
|------------------------------|----------------------------|
| (a) <code>i*j/j</code> | (b) <code>i/j*j</code> |
| (c) <code>(--i)*j</code> | (d) <code>(i++)*j</code> |
| (e) <code>(int)a * 10</code> | (f) <code>int(a*10)</code> |
| (g) <code>(i+a)*j</code> | (h) <code>(i+j)*a;</code> |

4. Compute the values for `ans1` and `ans2` from the following program:

```
main(void)
{
    float  ans1, data3;
    int    ans2, data1, data2;

    data1 = 3;
    data2 = 5;
    data3 = 3.5;
    ans1 = data2 / data1;
    ans2 = (3.3 * ((int) (data1 + data3)));
}
```



Branching

The execution of C programming statements is normally in sequence from the beginning to the end. From the previous chapters, we have discussed the simple data types, simple input/output, arithmetic calculations and simple assignment statements. With these statements, we can design simple C programs. However, the majority of challenging problems requires programs with the ability to make decisions as to what code to execute and the ability to execute certain portions of the code repeatedly. C provides a number of statements that allow branching and looping. This chapter discusses the branching structure that allows us to alter the normal sequential operations. The next chapter will discuss the looping structure that supports repetition.

This chapter covers the following topics:

- 6.1 Relational and Logical Operators
 - 6.2 The **if** Statement
 - 6.3 The **if-else** Statement
 - 6.4 The **if-else if-else** Statement
 - 6.5 The Nested-**if** Statement
 - 6.6 The **switch** Statement
 - 6.7 The Conditional Operator
-

6.1 Relational and Logical Operators

In a branching operation, the decision on which statements to be executed is based on a comparison between two values. To support this, C provides relational operators. Relational expressions involving relational operators are an essential

part of control structures for branching and looping. Relational operators in C are listed in Table 6.1. These operators are binary. They perform computations on their operands and return the result as either true or false. If the result is true, an integer value of 1 is returned, and if the result is false, then the integer value of 0 is returned.

Table 6.1 Relational operators.

Operator	Meaning	Example
<code>==</code>	equal to	<code>ch == 'a'</code>
<code>!=</code>	not equal to	<code>f != 0.0</code>
<code><</code>	less than	<code>num < 10</code>
<code><=</code>	less than or equal to	<code>num <= 10</code>
<code>></code>	greater than	<code>f > -5.0</code>
<code>>=</code>	greater than or equal to	<code>f >= 0.0</code>

Logical operators work on one or more relational expressions to yield either the logical value true or false. Table 6.2 lists the logical operators. Both logical **and** and logical **or** operators are binary operators, while the logical **not** operator is a unary operator. Logical operators allow testing and combining of the results of comparison expressions.

Table 6.2 Logical operators.

Operator	Meaning	Example
<code>!</code>	logical not	<code>!(num < 0)</code>
<code>&&</code>	logical and	<code>(num1 > num2) && (num2 > num3)</code>
<code> </code>	logical or	<code>(ch == '\t') (ch == ' ')</code>

Table 6.3 lists the results obtained when applying logical operators. Logical **not** operator (`!`) returns true when the operand is false and returns false when the operand is true. Logical **and** operator (`&&`) returns true when both operands are true, otherwise it returns false. Logical **or** operator (`||`) returns false when both operands are false, otherwise it returns true.

Table 6.3 Results of applying logical operators.

a	b	a && b	a b	!a	!b
true	true	true	true	false	false
true	false	false	true	false	true
false	true	false	true	true	false
false	false	false	false	true	true

Table 6.4 gives a list of operators of decreasing precedence. The logical **not**

(**!**) operator has the highest priority. It is followed by the multiplication, division, addition and subtraction operators. The logical **and** (**&&**) and **or** (**||**) operators have a lower priority than the relational operators. In general, any integer expression whose value is non-zero is considered true; otherwise it is false.

```
3 is true, 0 is false
1 && 0 is false
1 || 0 is true
!(5 >= 3) || (1) is true
```

Table 6.4 Operators with decreasing precedence.

Operator	Meaning
!	logical not
* /	multiply, divide
+ -	add, subtract
< <= > >=	less, less or equal, greater, greater or equal
== !=	equal, not equal
&&	logical and
 	logical or

The result of evaluating an expression involving relational and/or logical operators is either 1 or 0. When the result is true, it is 1. Otherwise it is 0, since C uses 0 to represent a false condition. An example is given in Program 6.1 to show the logic values of relational and logical expressions. The variable **result** is defined as type **float**. When the variable is printed with the specifier "**%f**", the true value to be printed will be 1.000000, and the false value to be printed will be 0.000000. The results are straightforward except the two special cases involving the evaluation of expressions consisting of logical **or** and logical **and** operators. First, in the evaluation of the following statement involving logical **and** operator:

```
result = (7 < 3) && (7/0 <= 5);
```

When the first relational expression is evaluated to be false, the second relational expression does not need to be evaluated. Therefore, even though the second expression contains an error in the evaluation of **7/0**, it does not occur in the overall result. Another case is in the evaluation of expressions containing logical **or** operator:

```
result = (4 == 4) || (32/0 == 0);
```

If the first expression is evaluated to be true, then the second expression does not need to be evaluated. In this case, although the second expression contains an error

when evaluating `32/0`, it is not evaluated. Hence, the error does not occur in the overall result.

Program 6.1 Evaluating relational and logical expressions.

```
#include <stdio.h>
main(void)
{
    float result;

    printf("The results of the logic relations:\n");
    result = (3 > 7);
    printf("(3 > 7) is %f\n", result);

    result = (7 < 3) && (3 <= 7);
    printf("(7 < 3) && (3 <= 7) is %f\n", result);
    result = (7 < 3) && (7/0 <= 5);
    printf("(7 < 3) && (7/0 <= 5) is %f\n", result);

    result = (32/4 > 3*4) || (4 == 4);
    printf("(32/4 > 3*4) || (4 == 4) is %f\n", result);
    result = (4 == 4) || (32/0 == 0);
    printf("(4 == 4) || (32/0 == 0) is %f\n", result);
    return 0;
}
```

Program output

```
The results of the logic relations:
(3 > 7) is 0.000000
(7 < 3) && (3 <= 7) is 0.000000
(7 < 3) && (7/0 <= 5) is 0.000000
(32/4 > 3*4) || (4 == 4) is 1.000000
(4 == 4) || (32/0 == 0) is 1.000000
```

6.2 The if Statement

The C language has two types of statements for implementing the branching control structure: **if** statement and **switch** statement. The **if** statement can be used for two-way selection, while the nested-**if** statement can be used for multi-way selection. Similarly, the **switch** statement is also used for multi-way selection.

The simplest form of the **if** statement is

```
if (expression)
    statement;
```

where **if** is a reserved keyword. If the **expression** is evaluated to be true (i.e. non-zero), then the **statement** is executed. If the **expression** is evaluated to be false (i.e. zero), then the **statement** is ignored, and the control is passed to the next program statement following the **if** statement. The **statement** may be a single statement terminated by a semicolon or a compound statement enclosed by braces. If the **statement** is a compound statement, then we have

```
if (expression) {
    statement1;
    statement2;
    ....
    statementn;
}
```

The flowchart for the **if** statement is given in Figure 6.1.

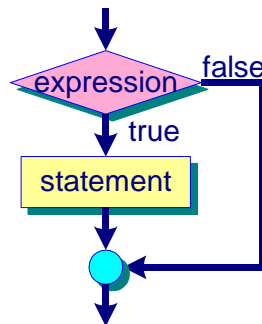


Figure 6.1 Flowchart of the if statement.

Program 6.2 gives a sample program using the **if** statement. The program asks the user to enter his examination mark and then reads the user input. The expression in the **if** statement contains a relational operator. It checks to see whether the user input is greater than or equal to 80 marks. If the relational expression is false, then the subsequent **printf()** statement is not executed. Otherwise, the **printf()** statement will print the string:

```
You score an A in your examination.
```

on the screen. Finally, the last **printf()** statement will be executed to print the mark entered by the user on the screen.

Program 6.2 Using the if statement.

```
#include <stdio.h>
main(void)
{
    int mark;

    printf("Give me your examination mark => ");
    scanf("%d", &mark);
    if (mark >= 80)
        printf("You score an A in your examination.\n");
    printf("Your mark is %d. \n", mark);
    return 0;
}
```

Program input and output

```
Give me your examination mark => 50
Your mark is 50.
```

```
Give me your examination mark => 88
You score an A in your examination.
Your mark is 88.
```

6.3 The if-else Statement

The **if-else** statement implements a two-way selection. The format of the **if-else** statement is

```
if (expression)
    statement1;
else
    statement2;
```

where **if** and **else** are reserved keywords. When the **if-else** statement is executed, the **expression** is evaluated. If **expression** is true, then **statement1** is executed and the control is passed to the program statement following the **if** statement. If **expression** is false, then **statement2** is executed. Both **statement1** and **statement2** may be a single statement terminated by a semicolon or a compound statement enclosed by **{}**. The flowchart for the **if-else** statement is given in Figure 6.2.

Program 6.3 computes the maximum number of two input integers. The two input integers are read in and stored in the variables **num1** and **num2**. The **if**

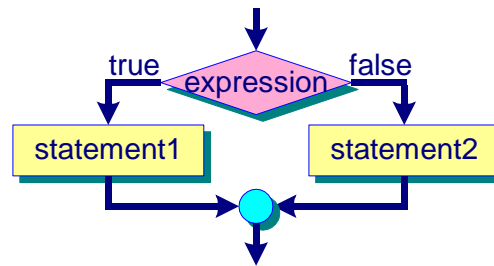


Figure 6.2 Flowchart of if-else statement.

statement is then used to compare the two variables, if **num1** is greater than **num2**, then the variable **max** is assigned with the value of **num1**. Otherwise, **max** is assigned with the value of **num2**. Figure 6.3 shows the flowchart for the program.

Program 6.3 Computing the maximum value of two input integers.

```

main(void)
{
    int num1, num2, max;

    printf("Enter two integers: ");
    scanf("%d %d", &num1, &num2);
    if (num1 > num2)
        max = num1;
    else
        max = num2;
    printf("The maximum of the two integers is %d.\n", max);
    return 0;
}
  
```

Program input and output

```

Enter two integers: 9 4
The maximum of the two integers is 9.

Enter two integers: -2 0
The maximum of the two integers is 0.
  
```

6.4 The if-else if-else Statement

The format for **if-else if-else** statement is

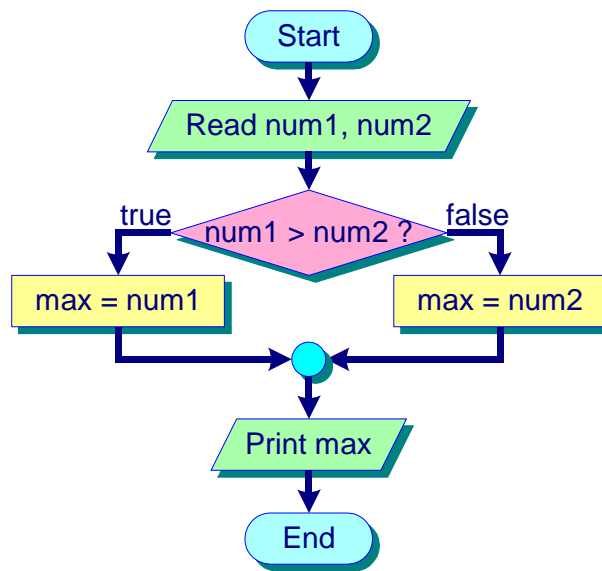


Figure 6.3 Flowchart of Program 6.3.

```
if (expression1)
    statement1;
else if (expression2)
    statement2;
else
    statement3;
```

each of **statement1**, **statement2** and **statement3** can either be a single statement terminated by a semicolon or a compound statement enclosed by **{}**. The flowchart for the **if-else if-else** statement is given in Figure 6.4.

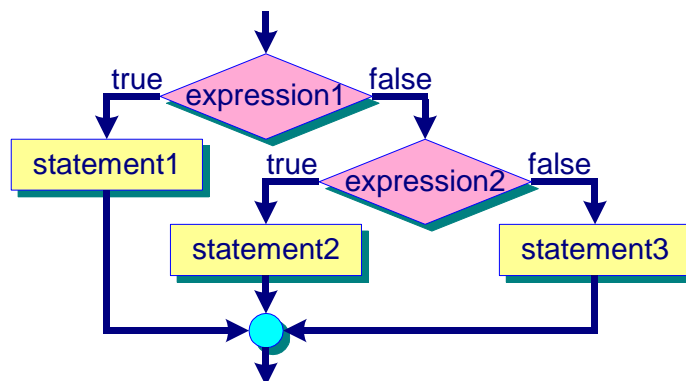


Figure 6.4 Flowchart of the if-else if-else statement.

We may have as many **else if** parts as possible in the **if** statement provided it is within the compiler limit:

```

if (expression1)
    statement1;
else if (expression2)
    statement2;
else if (expression3)
    statement3;
....
else
    statementN;

```

where **statementI** is executed when the first **I-1** expressions are false and the **expressionI** is true. If all the statements are false, then **statementN** will be executed. In any cases, only one statement will be executed, and the rest will be skipped. The last **else** part is optional and can be omitted. If the last **else** part is omitted, then no statement will be executed if all the expressions are evaluated to be false.

An example is given in Program 6.4 to illustrate the use of the **if-else if-else** statement. Figure 6.5 gives the flowchart for the program.

Program 6.4 Using the if-else if-else statement.

```

#include <stdio.h>
main(void)
{
    int mark;
    char grade;

    printf("Give me your examination mark => ");
    scanf("%d", &mark);
    if (mark <= 100 && mark >= 80)
        grade = 'A';
    else if (mark < 80 && mark >= 70)
        grade = 'B';
    else if (mark < 70 && mark >= 60)
        grade = 'C';
    else
        grade = 'F';

    printf("You score an %c in your examination.\n", grade);
    return 0;
}

```

Program input and output

Give me your examination mark => 85
 You score an A in your examination.

Give me your examination mark => 65
 You score an C in your examination.

Give me your examination mark => 35
 You score an F in your examination.

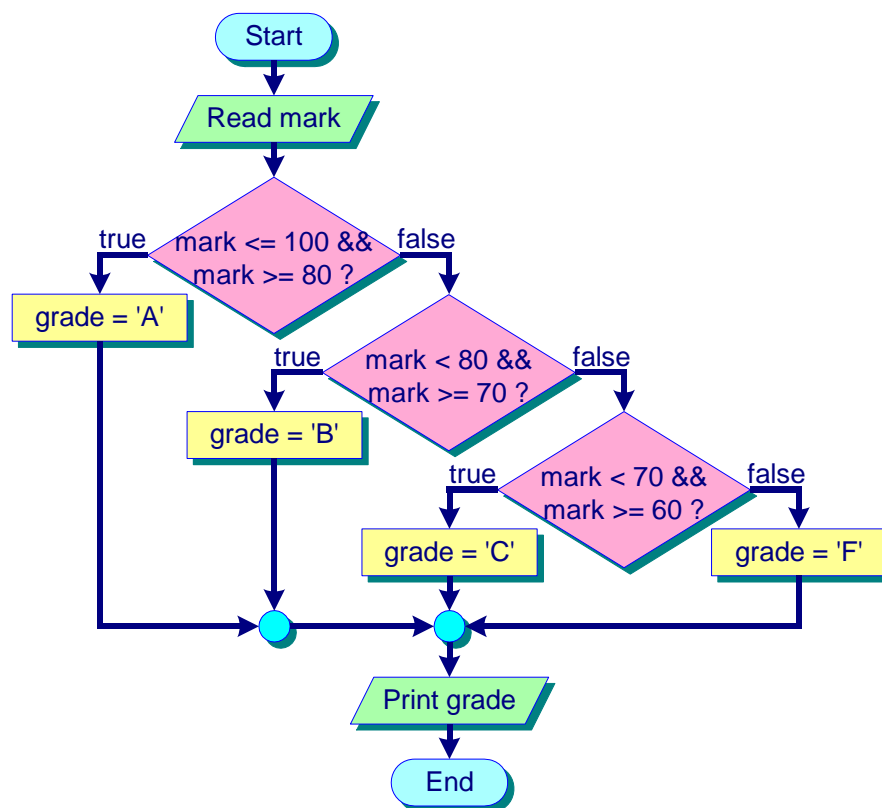


Figure 6.5 Flowchart for Program 6.4.

6.5 Nested-if

The nested-**if** statement allows us to perform a multi-way selection. In a nested-**if** statement, both the **if** branch and the **else** branch may contain one or more **if** statements. The level of nested-**if** statements can be as many as the limit the compiler allows. An example of a nested-**if** statement is

```
if (expression1)
    statement1;
else
    if (expression2)
        statement2;
    else
        statement3;
```

If **expression1** is true, then **statement1** is executed. If **expression1** is false and **expression2** is true, then **statement2** is executed. If both **expression1** and **expression2** are false, then **statement3** is executed. The flowchart of the nested-**if** structure is the same as Figure 6.4.

Another format of the nested-**if** statement is

```
if (expression1)
    if (expression2)
        statement1;
    else
        statement2;
else
    statement3;
```

If **expression1** and **expression2** are true, then **statement1** is executed. If **expression1** is true and **expression2** is false, then **statement2** is executed. If **expression1** is false, then **statement3** is executed. The flowchart is shown in Figure 6.6.

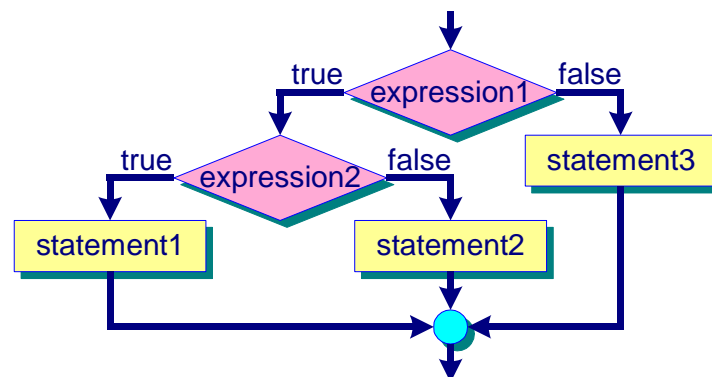


Figure 6.6 Flowchart for the nested-if statement.

A nested-**if** statement can cause confusion in some cases. For example, in the

following nested-**if** statement:

```
if (expression1)
    if (expression2)
        statement1;
    else
        statement2;
```

it is unclear to which **if** statement the **else** part belongs. C compiler associates an **else** part with the nearest unresolved **if**, i.e. the **if** statement that does not have an **else** statement. In this case, **statement1** is executed when **expression1** and **expression2** are true. **statement2** is executed when **expression1** is true and **expression2** is false. When **expression1** is false, neither of the two statements is executed. Thus, in this example, the **else** part is associated with the second **if** statement. To associate the **else** part with the first **if** statement, we can modify the statements using braces:

```
if (expression1) {
    if (expression2)
        statement1;
}
else
    statement2;
```

Program 6.5 illustrates the use of nested-**if** statements in computing the maximum value of three integers. This program reads in three integers from the user and stores the values in the variables **n1**, **n2** and **n3**. The values stored in **n1** and **n2** are then compared. If **n1** is greater than **n2**, then **n1** is compared with **n3**. If **n1** is greater than **n3**, then **max** is assigned with the value of **n1**, otherwise, **max** is assigned to the value of **n2**. On the other hand, if **n1** is less than **n2**, then **n2** is compared with **n3** in a similar way. The variable **max** is assigned with the value based on the comparison between **n2** and **n3**. Figure 6.7 shows the flowchart for Program 6.5.

Program 6.5 Computing the maximum value of three integers.

```
#include <stdio.h>
main(void)
{
    int n1, n2, n3, max;

    printf("Enter three integers: ");
    scanf("%d %d %d", &n1, &n2, &n3);
```

```

if (n1 >= n2) {
    if (n1 >= n3)
        max = n1;
    else
        max = n3;
}
else if (n2 >= n3)
    max = n2;
else
    max = n3;
printf("The maximum of the three integers is %d\n", max);
return 0;
}

```

Program input and output

Enter three integers: 1 2 3
 The maximum of the three integers is 3

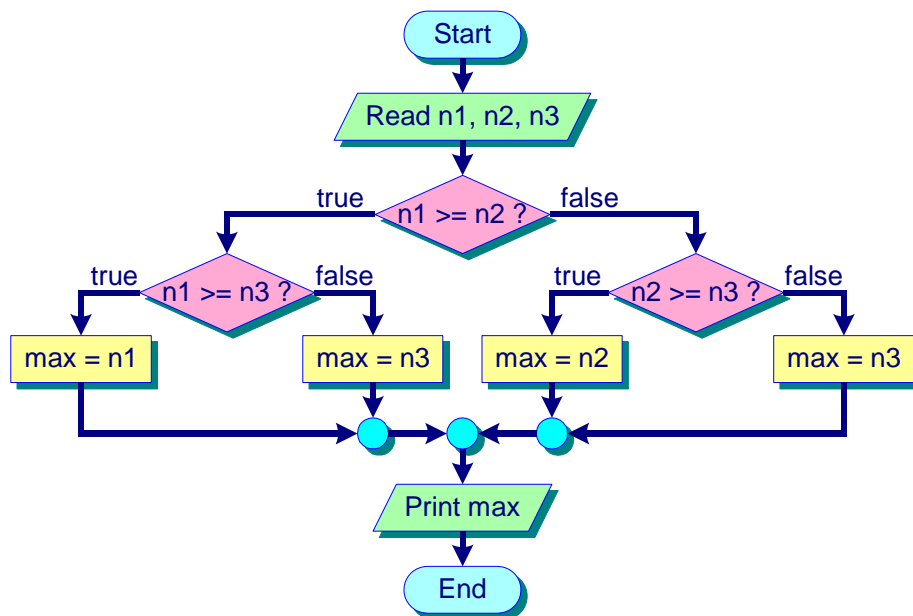


Figure 6.7 Flowchart for Program 6.5.

6.6 The switch Statement

The **switch** statement provides a multi-way decision structure in which one of

the several statements is executed depending on the value of an expression. The syntax of a **switch** statement is

```
switch (expression) {
    case constant1:
        statement1;
        break;
    case constant2:
        statement2;
        break;
    case constant3:
        statement3;
        break;
    ....
    default:
        statementD;
}
```

where **switch**, **case**, **break** and **default** are reserved keywords. **constant1**, **constant2**, etc. are called *labels*. Each must be an integer constant, a character constant or an integer constant expression, e.g. 3, 'A', 4+'b', 5+7, etc. Each of the labels must deliver a unique integer value. Duplicates are not allowed.

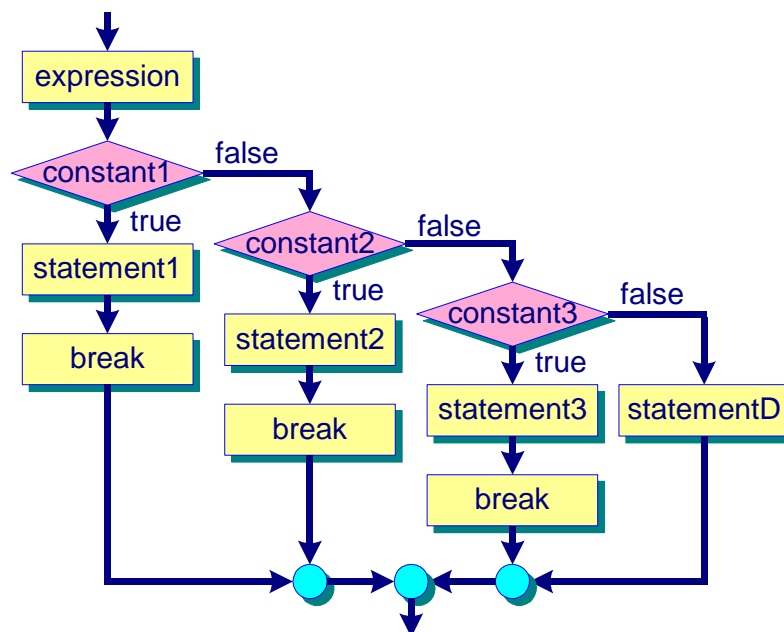


Figure 6.8 Flowchart of switch statement.

In the **switch** statement, **statementI** is executed only when the

expression has the value **constantI**. The **default** part is optional. If it is there, **statementD** is executed when **expression** has a value different from all the values specified by all the cases. The **break** statement signals the end of a particular case and causes the execution of the **switch** statement to be terminated. Each of the **statementI** may be a single statement terminated by a semicolon or a compound statement enclosed by **{ }**. Figure 6.8 shows the flowchart for the **switch** statement.

The **switch** statement is a better construct than multiple **if-else** statements. However, there are several restrictions in the use of the **switch** statement. The **expression** of the **switch** statement must return result of integer or character data type. The value in the **constantI** part must be an integer constant or character constant. Moreover, it does not support a range of values to be specified.

The **switch** statement is quite commonly used in menu-driven applications. Program 6.6 shows a program that uses the **switch** statement for menu-driven selection. The program displays a list of arithmetic operations that the user can enter. Then, the user selects the operation command and enters the two operands. The **switch** statement is then used to control which operation is to be executed based on user selection. The control is transferred to the appropriate branch of the **case** condition based on the variable **choice**, and the statements under the **case** condition are executed. In addition, we may also have multiple labels for a statement. As shown in Program 6.6, we also allow the choice to be specified in both lowercase and uppercase letters entered by the user. The flowchart for Program 6.6 is given in Figure 6.9.

Program 6.6 Using the switch statement.

```
#include <stdio.h>
main(void)
{
    char choice;
    int num1, num2, result;

    printf("Select an arithmetic operation:\n");
    printf("A-addition; S-subtraction; M-Multiplication\n");
    printf("Your choice (A, S or M) => ");
    scanf("%c", &choice);
    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);

    switch (choice)
    {
        case 'a':
        case 'A': result = num1 + num2;
```

```

        printf("%d + %d = %d\n", num1,num2,result);
        break;
    case 's':
    case 'S': result = num1 - num2;
        printf("%d - %d = %d\n", num1,num2,result);
        break;
    case 'm':
    case 'M': result = num1 * num2;
        printf("%d * %d2 = %d\n", num1,num2,result);
        break;
    default : printf("Not one of the proper choices.\n");
}
return 0;
}

```

Program input and output

```

Select an arithmetic operation:
A-addition; S-subtraction; M-Multiplication
Your choice (A, S or M) => s
Enter two numbers: 9 5
9 - 5 = 4

```

The **break** statement is used to end each **case** constant block statement. If we do not use the **break** statement in the **switch** statement, execution will continue with the statements for the subsequent **case** labels until a **break** statement or the end of the **switch** statement is reached. For example, if the **switch** statement is modified as follows:

```

switch(choice)
{
    case 'a':
    case 'A': result = num1 + num2;
        printf("%d + %d = %d\n", num1,num2,result);

    case 's':
    case 'S': result = num1 - num2;
        printf("%d - %d = %d\n", num1,num2,result);

    case 'm':
    case 'M': result = num1 * num2;
        printf("%d1 * %d = %d\n", num1,num2,result);
        break;
    default : printf("Not one of the proper choices.\n");
}

```

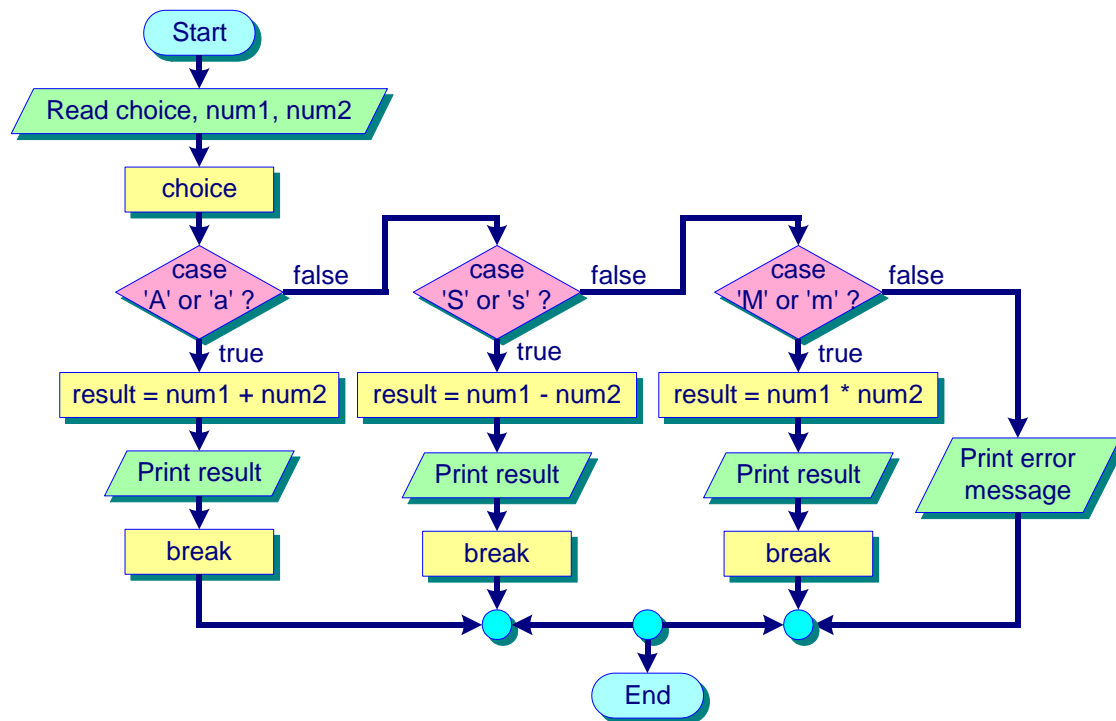


Figure 6.9 Flowchart for Program 6.6.

then if the user enters the choice 'A', the input and output from the program will become

```

Select an arithmetic operation:
A-addition; S-subtraction; M-Multiplication
Your choice (A, S or M) => A
Enter two numbers : 5 3
5 + 3 = 8
5 - 3 = 2
5 * 3 = 15
  
```

This is illustrated in Figure 6.10.

6.7 The Conditional Operator

The conditional operator is a ternary operator, which takes three expressions, with the first two expressions separated by a '?' and the second and third expressions separated by a ':'. The conditional operator is specified in the following way:

```
expression_1 ? expression_2 : expression_3
```

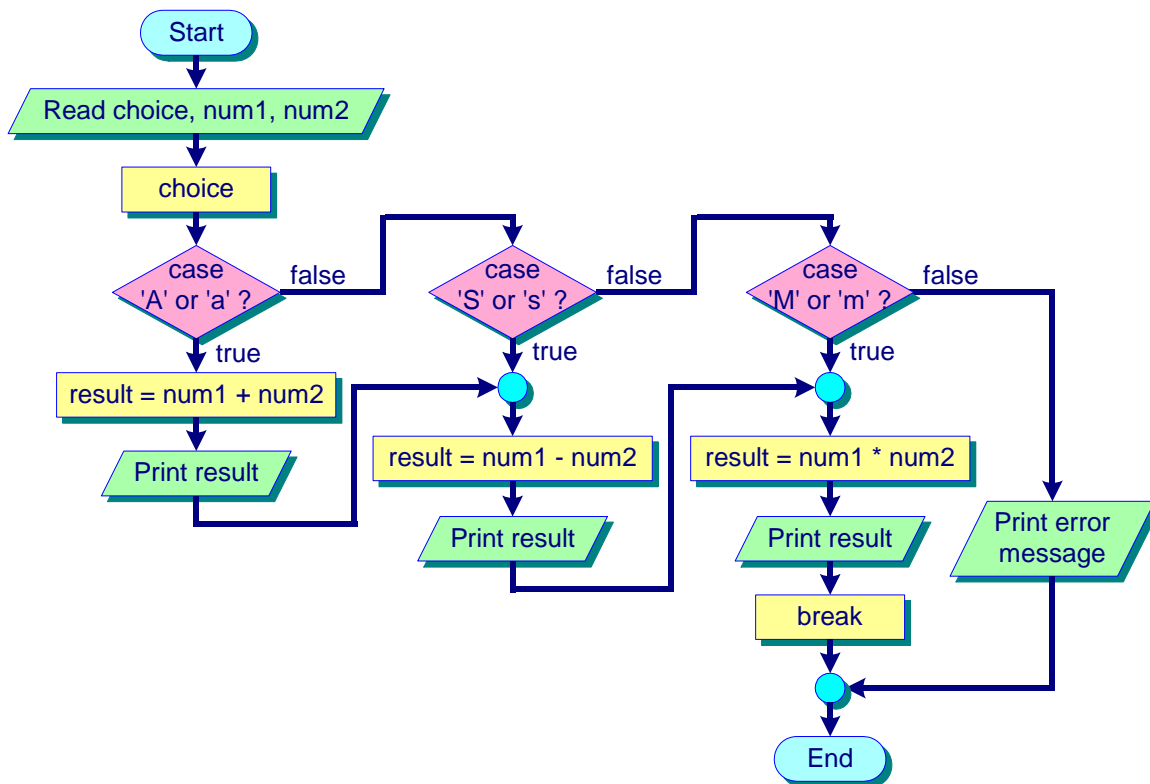


Figure 6.10 Flowchart for switch without the break statement.

The value of this expression depends on whether **expression_1** is true or false. If **expression_1** is true, the value of the expression becomes the value of **expression_2**, otherwise it is **expression_3**. The conditional operator is commonly used in an assignment statement, which assigns one of the two values to a variable. For example, the maximum value of the two values **x** and **y** can be obtained using the following statement:

```
max = x > y ? x : y;
```

The assignment statement is equivalent to the following **if-else** statement:

```
if (x > y)
    max = x;
else
    max = y;
```

Program 6.7 gives an example on the use of the conditional operator. The program reads a user input on the variable **choice**. If **choice** is 1, then the string "You have entered one." is printed. Otherwise, the string "You have

entered zero." is printed.

Program 6.7 Conditional operator.

```
#include <stdio.h>
main(void)
{
    int choice;

    printf("Enter your input (1 or 0) => ");
    scanf("%d", &choice);
    choice ? printf("You have entered one.\n") :
            printf("You have entered zero.\n");
    return 0;
}
```

Program input and output

Enter your input (1 or 0) => 1
You have entered one.

Enter your input (1 or 0) => 0
You have entered zero.

6.8 Exercises

1. Given the following declaration:

```
int x=1, y=2, z=3;
```

What are the values of the following expressions?

- (a) `x && y && z`
- (b) `x || y && z`
- (c) `x && y || z`
- (d) `x || !y && z`
- (e) `x-- <= y`
- (f) `x < y && z`

2. What is the output of the following code segment?

```
int x=1, y=2, z=0;
if (!(!x && !y) || z) {
    printf("First Statement");
}
```

```
else
    printf("Second Statement");
```

3. Given 4 integer variables, a1, a2, a3 and a4, write a program code to find the biggest value and the smallest value. The program code should have less than 5 comparisons.
4. Write a program to read input from the user and print a message to state whether the character is an uppercase letter, a lowercase letter or a digit.
5. Convert the following if statement to a switch statement. number is a variable with a value from 0 to 100.

```
if (number >= 75)
    printf("%c\n", 'A');
else if (number >= 65)
    printf("%c\n", 'B');
else if (number >= 55)
    printf("%c\n", 'C');
else if (number >= 45)
    printf("%c\n", 'D');
else
    printf("%c\n", 'F');
```



Looping

The branching constructs such as the **if-else** statement and the **switch** statement enable us make selection. Another kind of control structure is loop, which lets us execute a group of statements repeatedly for any number of times. There are three types of looping statements: **while** loop, **for** loop and **do-while** loop. In addition, the **break** statement and **continue** statement will also be discussed. The **break** statement can alter the control flow inside the **switch** statement and looping constructs such as **while**, **for** and **do-while** to cause immediate exit from the loop. The **continue** statement can only alter the flow inside a loop by skipping the remaining statements of the loop and go to the next iteration. The **goto** statement produces bad programming style and makes proving the correctness of the program difficult. **goto** will not be discussed.

This chapter covers the following topics:

- 7.1 Types of Loops
- 7.2 The **while** Statement
- 7.3 The **for** Statement
- 7.4 The **do-while** Statement
- 7.5 The **break** Statement
- 7.6 The **continue** Statement
- 7.7 Nested Loops

7.1 Types of Loops

Basically, there are two types of loops: *counter-controlled loops* and *sentinel-controlled loops*. In a counter-controlled loop, the loop body is repeated for a

specified number of times, and the number of repetitions is known before the loop begins execution. In a sentinel-controlled loop, the number of repetitions is not known before the loop begins execution. A loop control variable is typically used to determine the number of repetitions. The control variable is updated every time the loop is executed, and it is then tested in a test condition to determine whether to execute the loop body. An example of a *sentinel value* is a user input value such as `-1`, which should be different from regular data entered by the user.

To construct loops, we need the following four basic steps:

1. **Initialize.** This defines and initializes the loop control variable, which is used to control the number of repetitions of the loop.
2. **Test.** This evaluates the **test** condition, if the test condition is true, then the loop body is executed, otherwise the loop is terminated. The **while** loop and **for** loop evaluate the test condition at the beginning of the loop, while the **do-while** loop evaluates the test condition at the end of the loop.
3. **Loop body.** The **loop body** is executed if the condition is evaluated to be true. It contains the actual actions to be carried out.
4. **Update.** The **test** condition typically involves the loop control variable that should be modified each time through the execution of the loop body. The loop control variable will go through the test condition to determine whether to repeat the loop body again.

We can use one of the three looping constructs, namely the **while** loop, the **for** loop and the **do-while** loop, for constructing loops. However, not all the looping constructs contain the four steps in its declarations. For instance, the **while** loop contains only the **test** and **loop body** in its structure. **Update** needs to be written as part of the **loop body**, and **initialize** needs to be written explicitly before the loop starts. In contrast, the **for** loop contains the four basic steps in its declaration structure.

7.2 The while Statement

The format of the **while** statement is

```
while (test)  
    statement;
```

where **while** is a reserved keyword. **statement** can be a simple statement terminated by a semicolon or a compound statement enclosed by `{ }`. The **while** statement is executed by evaluating the **test** condition. If the result is true, then **statement** is executed. Control is then transferred back to the beginning of the **while** statement, and the process repeats again. This looping continues until the

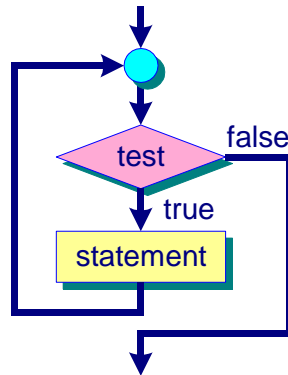


Figure 7.1 Flowchart for the while loop.

test condition finally becomes false. When the **test** condition is false, the loop terminates and the program continues executing the next sequential statement. The flowchart for the **while** loop is given in Figure 7.1.

The **while** loop is best to be used as a sentinel-controlled loop in situations where the number of times the loop to be repeated is not known in advance. For example, in the following code:

```

int mark=0, total=0;    /* initialize */
while (mark != -1) {    /* test */
    printf("Enter mark: (-1 to end the input)");
                        /* loop body */
    scanf("%d", &mark); /* update */
    total += mark;
}
  
```

the execution of the loop body is determined by the loop control variable **mark**. The user enters the **mark** and the sentinel value of **-1** is used to signal the end of user input.

The **while** loop can also be used as a counter-controlled loop. In the following code:

```

int counter=0, mark, sum=0; /* initialize */
while (counter < 10) {      /* test */
    printf("Enter the mark: "); /* loop body */
    scanf("%d", &mark);
    sum += mark;
    counter++;              /* update */
}
  
```

the loop control variable **counter** is defined and initialized. The loop control variable is incremented by one at the end of the loop body. The loop body will be executed when the **test** condition is evaluated true, i.e. **counter** is less than 10. The **test** condition will be evaluated as false when the counter value becomes 10, and the execution of the loop will be terminated.

Another common use of the **while** loop is to check the user input to see if the loop body is to be repeated. This is another example of sentinel-controlled loop. For example, in the following code:

```
char reply='Y';
while (reply != 'N') {
    printf("Repeat the loop body? (Y or N): \n");
    reply = getchar();
}
```

the loop will be terminated when the user enters the character 'N'.

Program 7.1 gives an example on converting temperature from Fahrenheit (F) into Celsius (C) using the **while** loop. The program reads in the upper limit of the temperature conversion and stores it in the variable **templimit**. It then uses the **while** loop as a counter-controlled loop. The loop control variable **fahren** is used to control the number of repetitions in the loop. The variable **fahren** is initialized as 32.0. In the loop body, the temperature conversion formula is used to convert the temperature from Fahrenheit to Celsius. The loop control variable **fahren** is also incremented by 10 every time when the loop body is executed. The loop will stop when **fahren** is greater than **templimit**.

Program 7.1 Converting from Celsius to Fahrenheit (using while loop).

```
#include <stdio.h>
main(void)
{
    float fahren;
    float templimit;

    printf("Please enter the temperature conversion limit\n(F): ");
    scanf("%f", &templimit);
    printf("\tFahrenheit\tCelsius\n");
    printf("\t-----\t-----\n");
    fahren = 32.0;
    while (fahren<=templimit) {
        printf("\t%.1f\t\t%.1f\n", fahren, (fahren - 32.0) *
            5.0 / 9.0);
        fahren += 10;
    }
```

```

    }
    return 0;
}

```

Program input and output

Please enter the temperature conversion limit (F): 112.0

Fahrenheit	Celsius
-----	-----
32.0	0.0
42.0	5.6
52.0	11.1
62.0	16.7
72.0	22.2
82.0	27.8
92.0	33.3
102.0	38.9
112.0	44.4

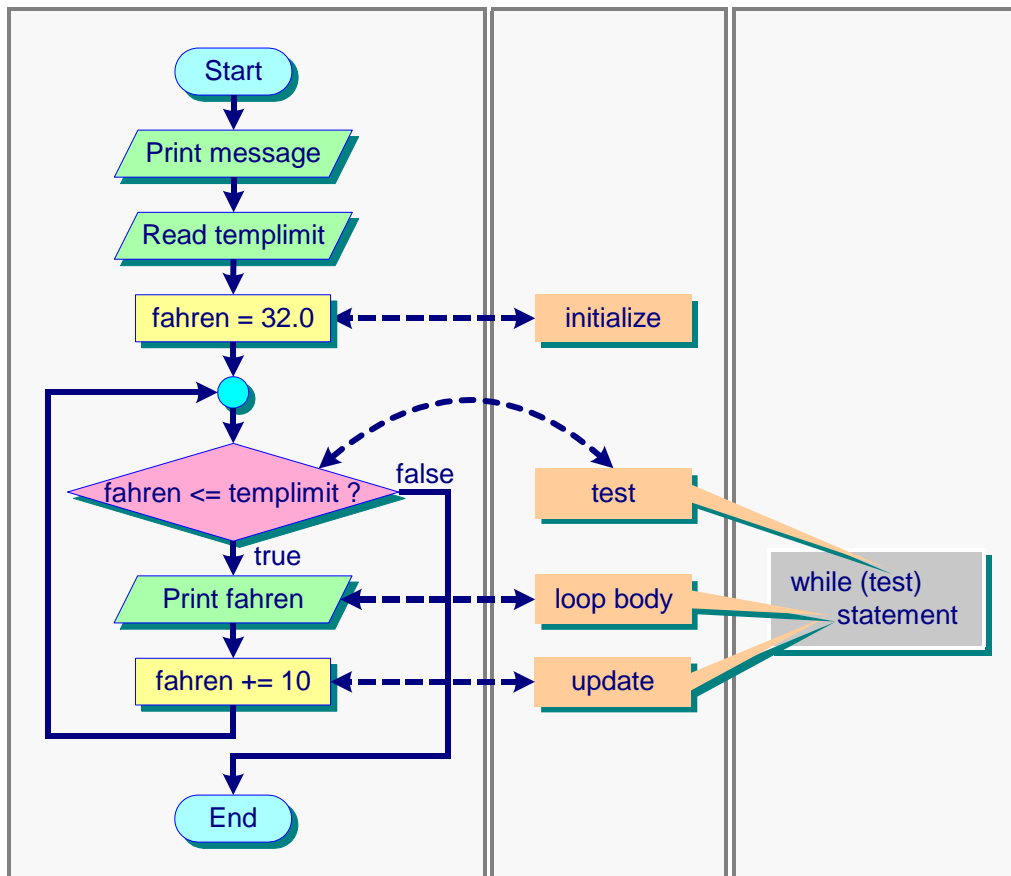


Figure 7.2 Flowchart for Program 7.1.

The flowchart for Program 7.1 is given in Figure 7.2. In the **while** loop, **initialize** has to be written explicitly, and **statement** includes both the **loop body** and **update**.

Program 7.2 gives an example on using the **while** loop. The program does not know how much input is to be read at the beginning of the program. However, it will keep on reading the data until the user input is -1 which is the sentinel value. The **scanf()** statement reads in the first number and stores it in the variable **item**. Then, the execution of the **while** loop begins. If the initial **item** value is not -1 , then the statements in the braces are executed. The **item** value is first added to another variable **sum**. Another **item** value is then read in from the user, and the control is transferred back to the **test** expression (**item** $\neq -1$) for evaluation. This process repeats until the **item** value becomes -1 . The flowchart for Program 7.2 is given in Figure 7.3.

Program 7.2 Calculating the sum of integers using while loop.

```
#include <stdio.h>
main(void)
{
    int  sum=0, item;

    printf("Enter the list of integers:\n");
    scanf("%d", &item);
    while (item != -1) {
        sum += item;
        scanf("%d", &item);
    }
    printf("The sum is %d\n", sum);
    return 0;
}
```

Program input and output

Enter the list of integers:

1 8 11 24 36 48 67 -1

The sum is 195

Please enter the list of integers:

-1

The sum is 0

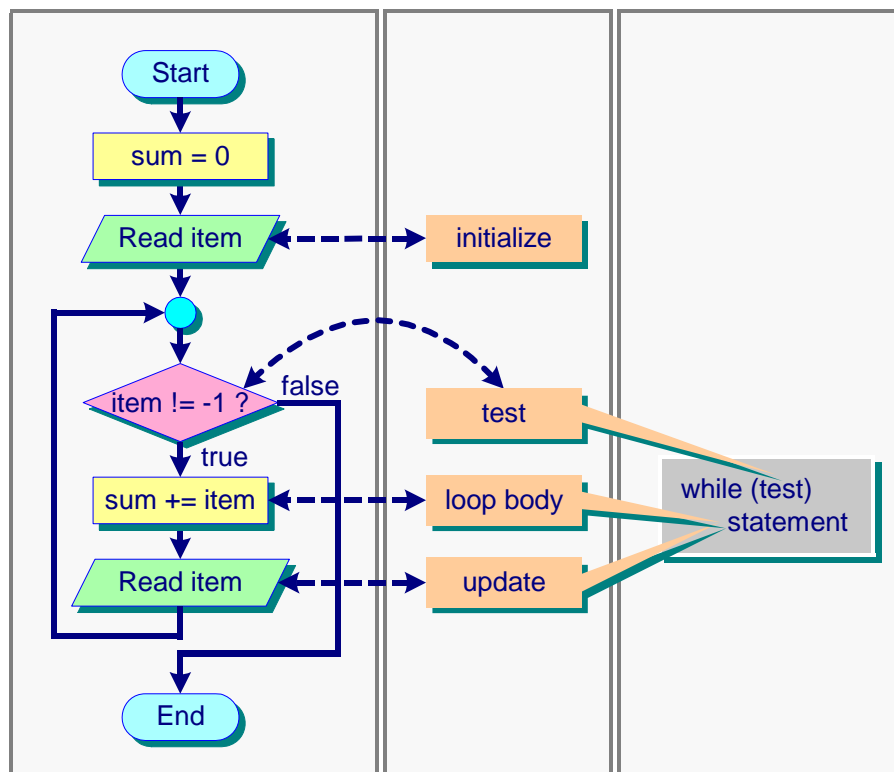


Figure 7.3 Flowchart for Program 7.2.

Program 7.3 finds x to the power of y (i.e. x^y). The program reads in two values and stores them in two variables x and y where x is of type `float` and y is of type `int`. The variable `power` is used to store the result, and it is initialized to 1.0. As the value of y can be positive or negative, the `if` statement is used to control the execution of statements based on the value of y . If x equals to zero, then the result is zero. If the value of y is negative, i.e. $y < 0$, then a `while` loop is used. The statements

```
while (y++)
    power *= 1/x;
```

are equivalent to the following statements

```
while (y == TRUE){
    power *= 1/x;
    y = y+1;
}
```

For example, if $x = 3$ and $y = -2$, then when the `while` loop is first executed, `power` = 1.0 * 1/3, and $y = -1$. The expression `(y == TRUE)` is true, so the loop is executed again. This time, `power` = 1.0 * 1/3 * 1/3, and $y = 0$. Now the `test`

expression (**y == TRUE**) is false as **y** equals to zero. The loop terminates and the result is 1/9, i.e. 0.111111.

Program 7.3 Finding the power of a number.

```
#include <stdio.h>

main(void)
{
    float x;
    int y;
    float power = 1.0;

    printf("Please enter x and y (as xy): ");
    scanf("%f %d", &x, &y);
    if (x == 0.0)
        power = 0.0;
    else if (y < 0)
        while (y++)
            power *= 1/x;
    else while (y--)
        power *= x;
    printf("Result is %8.3f\n", power);
    return 0;
}
```

Program input and output

```
Please enter x and y (as xy): 2 -3
Result is      0.125

Please enter x and y (as xy): 2 0
Result is      1.000

Please enter x and y (as xy): 2 4
Result is     16.000
```

If **y** is not less than zero, then another **while** loop is used. The statements

```
while (y--)  
    power *= x;
```

are equivalent to the following statements

```
while (y == TRUE){
```

```

    power *= x;
    y = y-1;
}

```

For example, if $x = 3$ and $y = 2$, then when the **while** loop is first executed, $\text{power} = 1.0 * 3$, and $y = 1$. The expression ($y == \text{TRUE}$) is true, so the loop is executed again. This time, $\text{power} = 1.0 * 3 * 3$, and $y = 0$. Now the **test** expression ($y == \text{TRUE}$) is false as y equals to zero. The loop terminates and the result is 9.0.

Figure 7.4 shows the flowchart for the operation of Program 7.3 when $x = 3$ and $y = 2$.

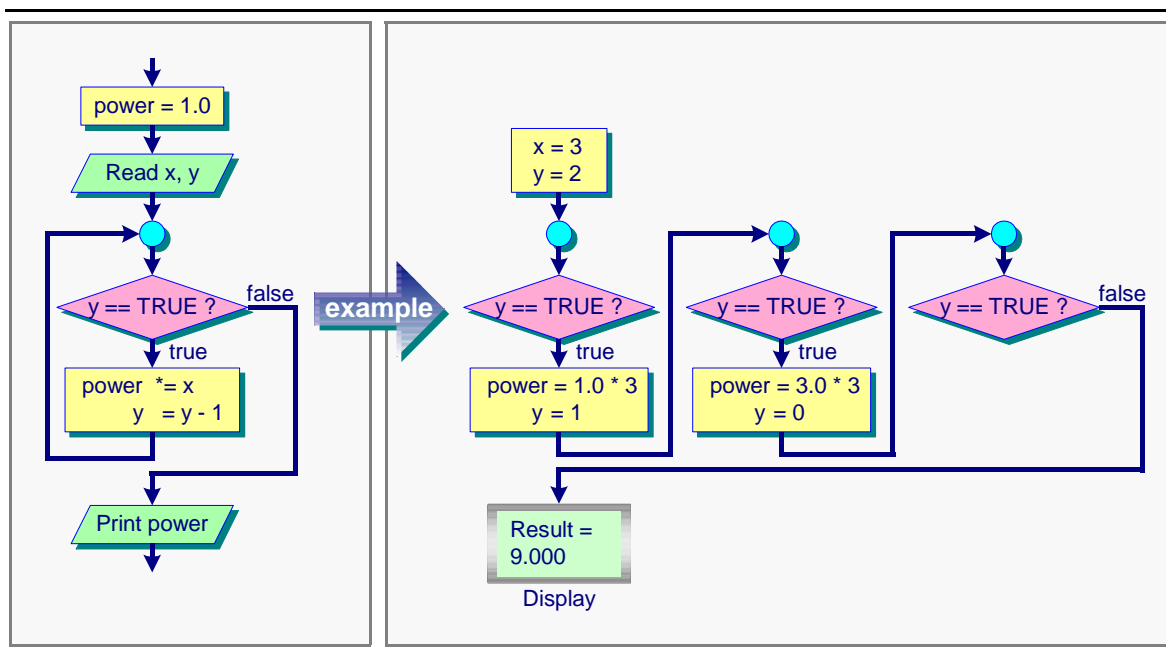


Figure 7.4 Flowchart for Program 7.3.

7.3 The for Statement

The **for** statement allows us to repeat a sequence of statements for a specified number of times which is known in advance. The **for** loop is mainly used as a *counter-controlled loop*. The format of the **for** statement is

```

for (initialize; test; update)
    statement;

```

where **for** is the reserved keyword. **statement** can be a simple statement terminated by a semicolon or a compound statement enclosed by `{}`. **initialize** is usually used to set the initial value of one or more loop control

variables. Normally, **test** is a relational expression to control iterations. **update** is used to update some loop control variables before repeating the loop. The flowchart for the **for** loop is given in Figure 7.5.

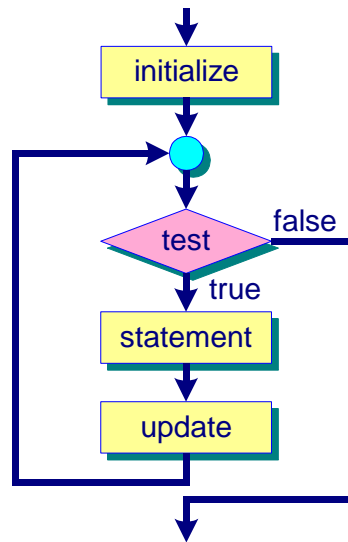


Figure 7.5 Flowchart for the for loop.

In the **for** loop, **initialize** is first evaluated. The **test** condition is then evaluated. If the **test** condition is true, then the **statement** and **update** expression are executed. Control is then transferred to the **test** condition, and the loop is repeated again if the **test** condition is true. If the **test** condition is false, then the loop is terminated. The control is then transferred out of the **for** loop to the next sequential statement.

The **for** statement can be represented by using a **while** statement as follows:

```
initialize;
while (test) {
    statement;
    update;
}
```

The difference is that in the **while** loop, we only specify the **test** condition and the **statement** in the loop, the **initialize** and **update** need to be added at the appropriate locations in order to make the **while** loop equivalent to the **for** loop.

The **for** loop is mainly used as a counter-controlled loop. For example, the following code


```
for (n=0; n<10; ++n) {
    sum += 5;
}
```

will add 5 to the variable **sum** every time the loop is executed. The number of time the loop is to be executed is known in advance. In this case, the loop will be executed 10 times. The loop will terminate when **n** becomes 10 and the **test** expression (**n<10**) becomes false. Consider the statements

```
for (n=100; n>10; n-=5) {
    total += 5;
}
```

the loop counts backward from 100 to 10. Each step will be decremented by 5. Therefore, the loop will be executed for a total of 18 times. For each time, the variable **n** will be decremented by 5, until it reaches 10.

Any or all of the 3 expressions may be omitted in the **for** loop. For example, the statements

```
for (n=5; n<=10 && n>=1;) {
    scanf("%d", &n);
}
```

are valid and the **update** expression is omitted. Notice that complex **test** conditions can be set using relational operators. The statements

```
for (; n<=10; ++n) {
    statement;
    ....
}
```

are also valid when the **initialize** expression is omitted.

In the case where the **test** expression is omitted, it becomes an infinite loop, i.e. all statements inside the loop will be executed again and again.

```
for (;;) {          /* an infinite loop */
    statement;
    ....
}
```

Notice that the semicolons must be included even if the expression is omitted.

In addition, we can also use more than one expression in **initialize** and **update**. A comma operator (,) is used to separate the expressions:

```

    for (count=0, sum=0; count<5; count++) {
        sum+=count;
    }

```

The **for** statement has two expressions in **initialize**. We can also have the **for** statement to perform the above task as follows:

```

    for (count=0, sum=0; count<5; sum+=count, count++)
        ;

```

For the **for** loop, the loop body is null (;) which does nothing.

Program 7.4 Converting from Celsius to Fahrenheit (using for loop).

```

#include <stdio.h>
main(void)
{
    float fahren;
    float templimit;

    printf("Please enter the temperature conversion limit
           (F): ");
    scanf("%f", &templimit);

    printf("\tFahrenheit\tCelsius\n");
    printf("\t-----\t-----\n");
    for (fahren=32.0; fahren<=templimit; fahren += 10)
        printf("\t%.1f\t\t%.1f\n", fahren, (fahren - 32.0) *
              5.0 / 9.0);
    return 0;
}

```

Program 7.4 gives an example on converting temperature from Fahrenheit (F) into Celsius (C) using the **for** loop. The program reads in the upper limit of the temperature conversion and stores it in the variable **templimit**. It then uses the **for** loop as a counter-controlled loop. The loop control variable **fahren** is used to control the number of repetitions in the loop. The variable **fahren** is initialized to 32.0. In the loop body, the temperature conversion formula is used to convert the temperature from Fahrenheit to Celsius. The loop control variable **fahren** is also incremented by 10 every time the loop body finishes executing. The loop will stop when **fahren** equals to **templimit**. The flowchart for Program 7.4 is given in Figure 7.6.

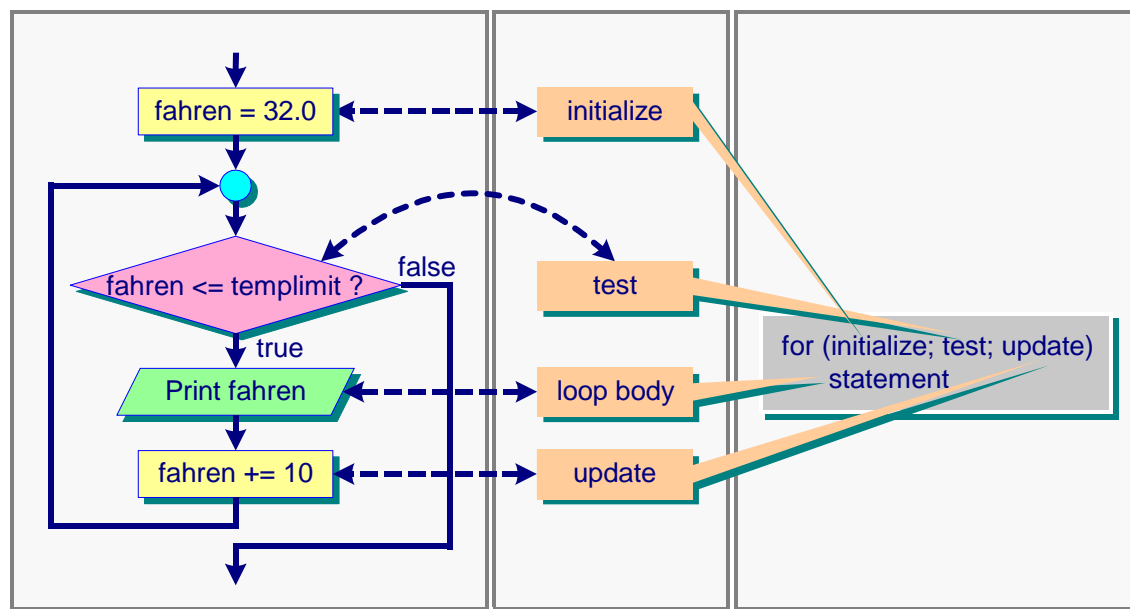


Figure 7.6 Flowchart for Program 7.4.

Another example of using the **for** loop is given in Program 7.5. The program calculates a series of data as follows:

$$1 - \frac{x}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \dots + \frac{x^{20}}{20!}$$

Program 7.5 Summing a series of data using for loop.

```

#include <stdio.h>
main(void)
{
    double x, temp=1.0, term=1.0;
    int n, sign=1, denom=1;

    printf("Please enter the value of x: ");
    scanf("%lf", &x);

    for (n=1; n<=20; n++) {
        denom *= n;
        sign = -sign;
        term *= x;
        temp += sign * term/denom;
    }
    printf("The result is %lf\n", temp);
    return 0;
}

```

Program input and output

Please enter the value of x: 1.2
The result is 0.301194

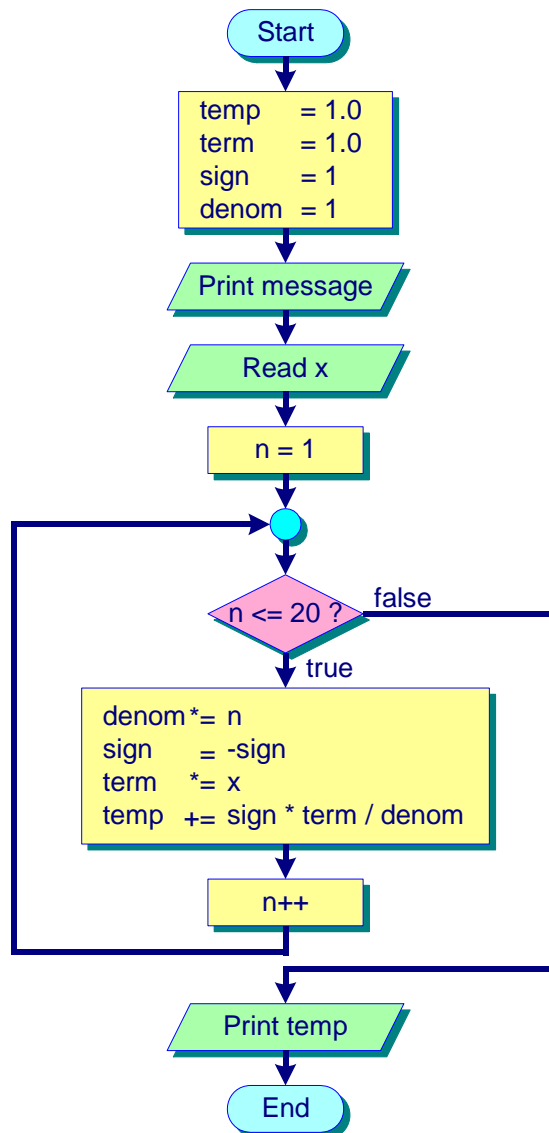


Figure 7.7 Flowchart for Program 7.5.

Before we write the program, we observe that each component of the series consists of three parts: the part involving **x**, the part on the factorial, and the sign. Therefore, we create three variables, namely **term**, **denom** and **sign** to store the

data of each part of the component. In addition, the variable **temp** is used to calculate the value for the series for each component of the **for** loop. The variable **temp** is initialized to 1.0. The loop control variable **n** is used to control the loop execution. It is initialized to 1. It stops execution after 20 iterations when **n** equals 21. The flowchart for Program 7.5 is given in Figure 7.7.

7.4 The do-while Statement

Both the **while** and the **for** loops test the condition prior to executing the loop body. C also provides the **do-while** statement which implements a control pattern different from the **while** and **for** loops. The body of a **do-while** loop is executed at least once. Its general format is

```
do
    statement;
while (test);
```

The **do-while** loop differs from the **while** and **for** statements in that the condition **test** is performed after the **statement** finishes executing every time. This means the loop will be executed at least once. On the other hand, the body of the **while** or **for** loop might not be executed even once. **statement** can be a simple statement terminated by a semicolon or a compound statement enclosed by **{ }**. Figure 7.8 gives the flowchart for the **do-while** loop.

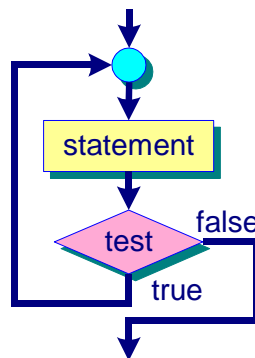


Figure 7.8 Flowchart for the do-while loop.

Program 7.6 gives an example on converting temperature from Fahrenheit (F) into Celsius (C) using the **do-while** loop. The **do-while** loop is used as a counter-controlled loop in this program. The loop control variable **fahren** is used to control the number of repetitions in the loop. The variable **fahren** is initialized

to 32.0. The loop control variable **fahren** is also incremented by 10 every time the loop body is executed. The loop will stop when **fahren** equals to **templimit**. The flowchart for Program 7.6 is given in Figure 7.9.

Program 7.6 Converting from Celsius to Fahrenheit (using do-while loop).

```
#include <stdio.h>
main(void)
{
    float fahren;
    float templimit;

    printf("Please enter the temperature conversion limit
        (F): ");
    scanf("%f", &templimit);
    printf("\tFahrenheit\tCelsius\n");
    printf("\t-----\t-----\n");
    fahren = 32.0;
    do {
        printf("\t%.1f\t\t%.1f\n", fahren, (fahren - 32.0) *
            5.0 / 9.0);
        fahren+=10;
    } while (fahren<=templimit);
    return 0;
}
```

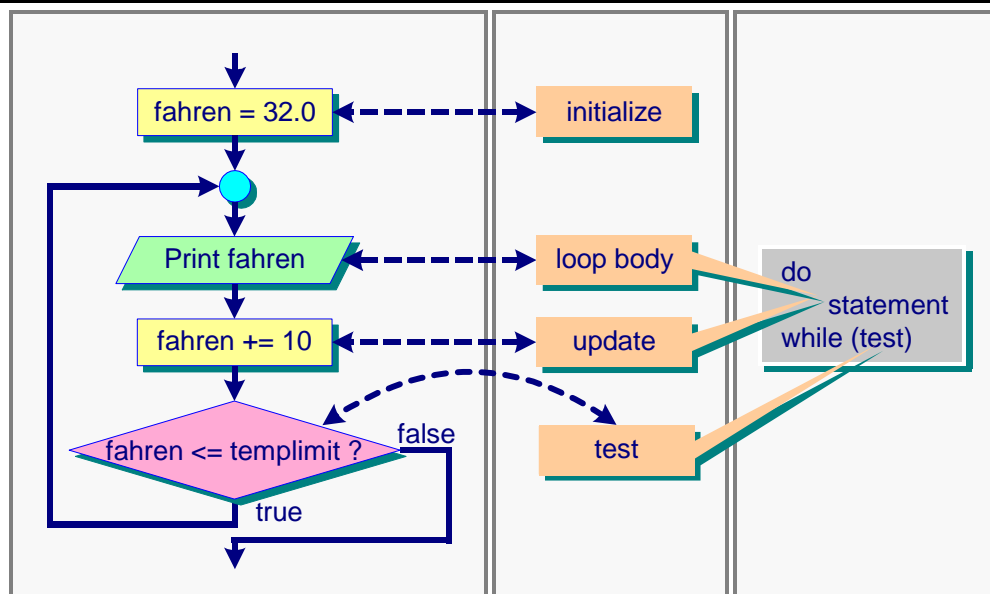


Figure 7.9 Flowchart for Program 7.6.

Program 7.7 gives an example on using the **do-while** statement. The program reads in a number between 1 and 5. If the number entered is not within the range, an error message is printed on the display to prompt the user to input the number again. The program will read the user input at least once.

Program 7.7 Using do-while loop.

```
#include <stdio.h>

main()
{
    int input; /* User input number. */

    do {
        printf("Input a number >= 1 and <= 5: ");
        scanf("%d",&input);
        if (input > 5 || input < 1)
            printf("%d is out of range, try again.\n", input);
    } while (input > 5 || input < 1);
    printf("Input = %d\n", input);
    return 0;
}
```

Program input and output

```
Input a number >= 1 and <= 5: 0
0 is out of range, try again.
Input a number >= 1 and <= 5: 6
6 is out of range, try again.
Input a number >= 1 and <= 5: 5
Input = 5
```

The **do-while** loop differs from the **while** loop in that the **while** loop evaluates the **test** condition at the beginning of the loop, whereas the **do-while** loop evaluates the **test** condition at the end of the loop. If the initial **test** condition is true, the two loops will have the same number of iterations. The number of iterations between the two loops will differ only when the initial **test** condition is false. In this case, the **while** loop will exit without executing any statements in the loop body. But the **do-while** loop will execute the loop body at least once before exiting from the loop. Therefore, the **do-while** statement is useful for situations where it requires executing the loop body at least once.

Both the **while** loop and **do-while** loop can be used as sentinel-controlled loops. They can be used in situations where we do not know the number of

iterations in advance. When it comes to choosing which looping statement to use, the **while** loop is generally preferred as it provides the extra control even on executing the loop body for the first time.

7.5 The break Statement

The **break** statement alters flow of control inside a **for**, **while** or **do-while** loop, as well as the **switch** statement. The execution of **break** causes immediate termination of the innermost enclosing loop or the **switch** statement. The control is then transferred to the next sequential statement following the loop.

The **break** statement is usually used in a special situation where immediate termination of the loop is required. This is shown in Program 7.8. This program computes the area of a rectangle. It reads in the length and width of the rectangle from the user. A **while** loop is used to read in user input repeatedly and calculates the area of the rectangle accordingly. The **break** statement is used in the **if** statement inside the **while** loop. The **if** statement will check the number of each user input on *width*. If it is not 1, then the **break** statement will be executed, which will then terminate the **while** loop and exit.

Program 7.8 Using the break statement.

```
#include <stdio.h>

main(void)
{
    float length, width;
    printf("Enter the length of the rectangle: ");
    while (scanf("%f", &length) == 1) {
        printf("Enter the width: ");
        if (scanf("%f", &width) != 1)
            break;
        printf("The area = %.1f\n", length * width);
        printf("Enter the length of the rectangle: ");
    }
    return 0;
}
```

Program input and output

```
Enter the length of the rectangle: 10
Enter the width: 20
The area = 200.0
Enter the length of the rectangle: 8
```


Enter the width: a

Figure 7.10 gives the flowchart for Program 7.8. In Figure 7.10, **num_input** represents the number of user inputs.

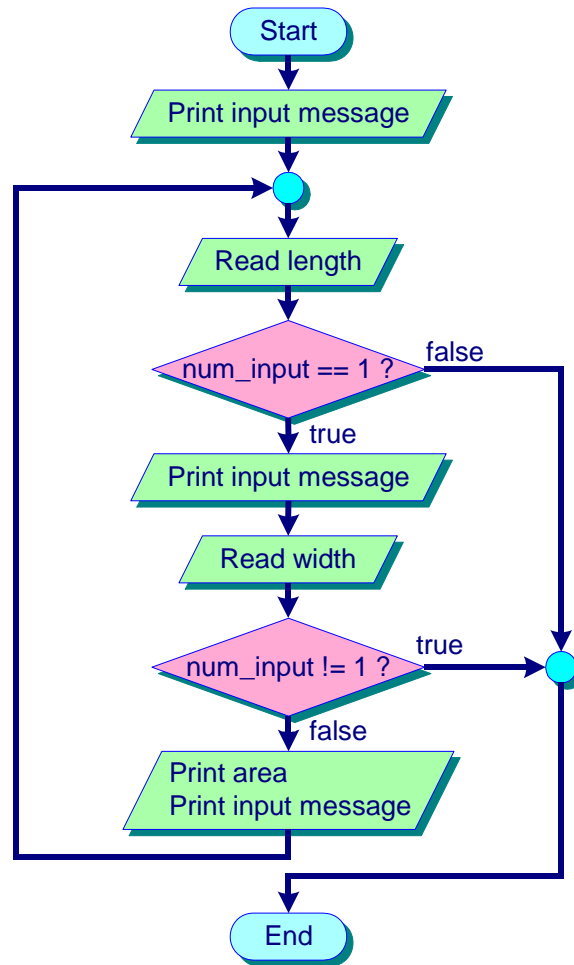


Figure 7.10 Flowchart for Program 7.8.

However, it is important to notice that if the **break** statement is executed inside a nested loop, the **break** statement will only terminate the innermost loop. In the following statements:

```

for (i=0; i<10; i++) {
    for (j=0; j<20; j++) {
        if (i == 3 || j == 5)
            break;
        else

```

```

        printf("Print the values %d and %d.\n", i, j);
    }
}

```

the **break** statement will only terminate the innermost loop of the **for** statement when **i** equals 3 or **j** equals 5. When this happens, the outer loop will carry on execution with **i** equals 4, and the inner loop starts execution again.

7.6 The continue Statement

The **continue** statement complements the **break** statement. Its use is restricted to the **while**, **for** and **do-while** loop. The **continue** statement causes termination of the current iteration of a loop and the control is immediately passed to the **test** condition of the nearest enclosing loop. All subsequent statements after the **continue** statement are not executed for this particular iteration.

The **continue** statement differs from the **break** statement in that the **continue** statement terminates the execution of the current iteration, and the loop still carries on with the next iteration if the **test** condition is fulfilled, while the **break** statement terminates the execution of the loop and passes the control to the next statement immediately after the loop. However, the use of the **break** statement and **continue** statement are discouraged in good programming practice, as both statements interrupt normal sequential execution of the program.

Program 7.9 gives an example on using the **continue** statement to sum up a list of positive numbers. The program reads eight numbers of data type **float**. A **for** loop is used to process the input number one by one. If the number is not less than zero, then the value is added to **sum**, otherwise the **continue** statement is used to terminate the current iteration of the loop, and the control is transferred to the next iteration of the loop. Notice that the **for** loop will process all the eight numbers each time when they are read. Figure 7.11 gives the flowchart for Program 7.9.

Program 7.9 Summing up a list of positive numbers.

```

#include <stdio.h>
main(void)
{
    int i;
    float data, sum=0;

    printf("Enter 8 numbers: ");
    /* read 8 numbers */
    for (i=0; i<8; i++) {

```

```
scanf("%f", &data);  
if (data < 0.0)  
    continue;  
sum += data;  
}  
printf("The sum is %f\n",sum);  
return 0;  
}
```

Program input and output

Enter 8 numbers: 1 2 3 4 -5 6 -7 8
The sum is 24.000000

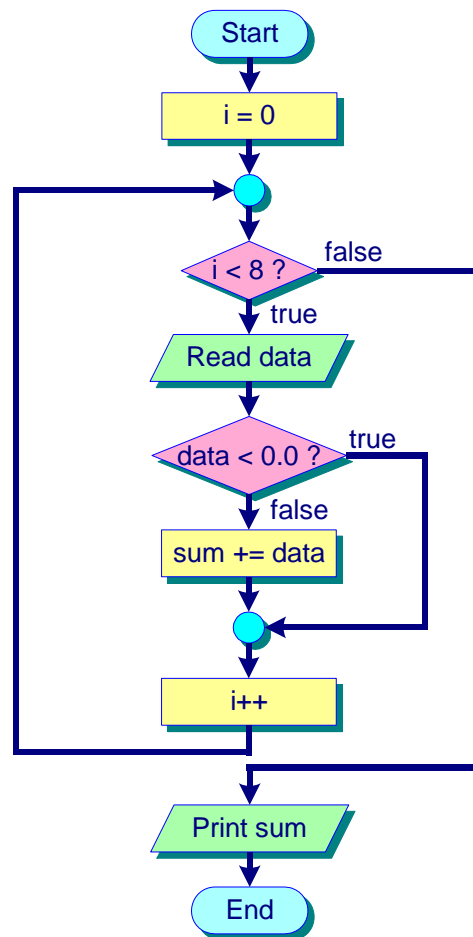


Figure 7.11 Flowchart for Program 7.9.

7.7 Nested Loops

A loop may appear inside another loop. This is called a *nested* loop. We can nest as many levels of loops as the system allows. We can also nest different types of loops.

Program 7.10 generates the different strings of letters from the characters 'a', 'b' and 'c'. The program contains a nested loop, in which one **for** loop is nested inside another **for** loop. The counter variables **i** and **j** are used to control the **for** loops. Another variable **num** is used as an overall counter to record the number of strings that have been generated by the different combinations of the characters. The nested loop is executed as follows. In the outer **for** loop, when the counter variable **i='a'**, the inner **for** loop is executed, and generates the different strings **aa**, **ab** and **ac**. When **i='b'** in the outer **for** loop, the inner **for** loop is executed and generates **ba**, **bb** and **bc**. Similarly, when **i='c'** in the outer **for** loop, the inner **for** loop generates **ca**, **cb** and **cc**. The nested loop is then terminated. The total number of strings generated is also printed on the screen. Figure 7.12 shows the flowchart for Program 7.10.

Program 7.10 Printing strings of characters using nested for loop.

```
#include <stdio.h>
main(void)
{
    char i,j;          /* for-loop counters */
    int num = 0;       /* overall loop counter */

    printf("The strings generated are:\n");
    for (i = 'a'; i <= 'c'; i++) {      /* outer loop */
        for (j = 'a'; j <= 'c'; j++){   /* inner loop */
            num++;
            printf("%c%c ",i,j);
        }                               /* end inner loop */
        printf("\n");
    }                                    /* end outer loop */
    printf("A total of %d different strings are formed.\n",
        num);
    return 0;
}
```

Program output

```
The strings generated are:
aa ab ac
ba bb bc
```

```
ca cb cc
```

A total of 9 different strings are formed.

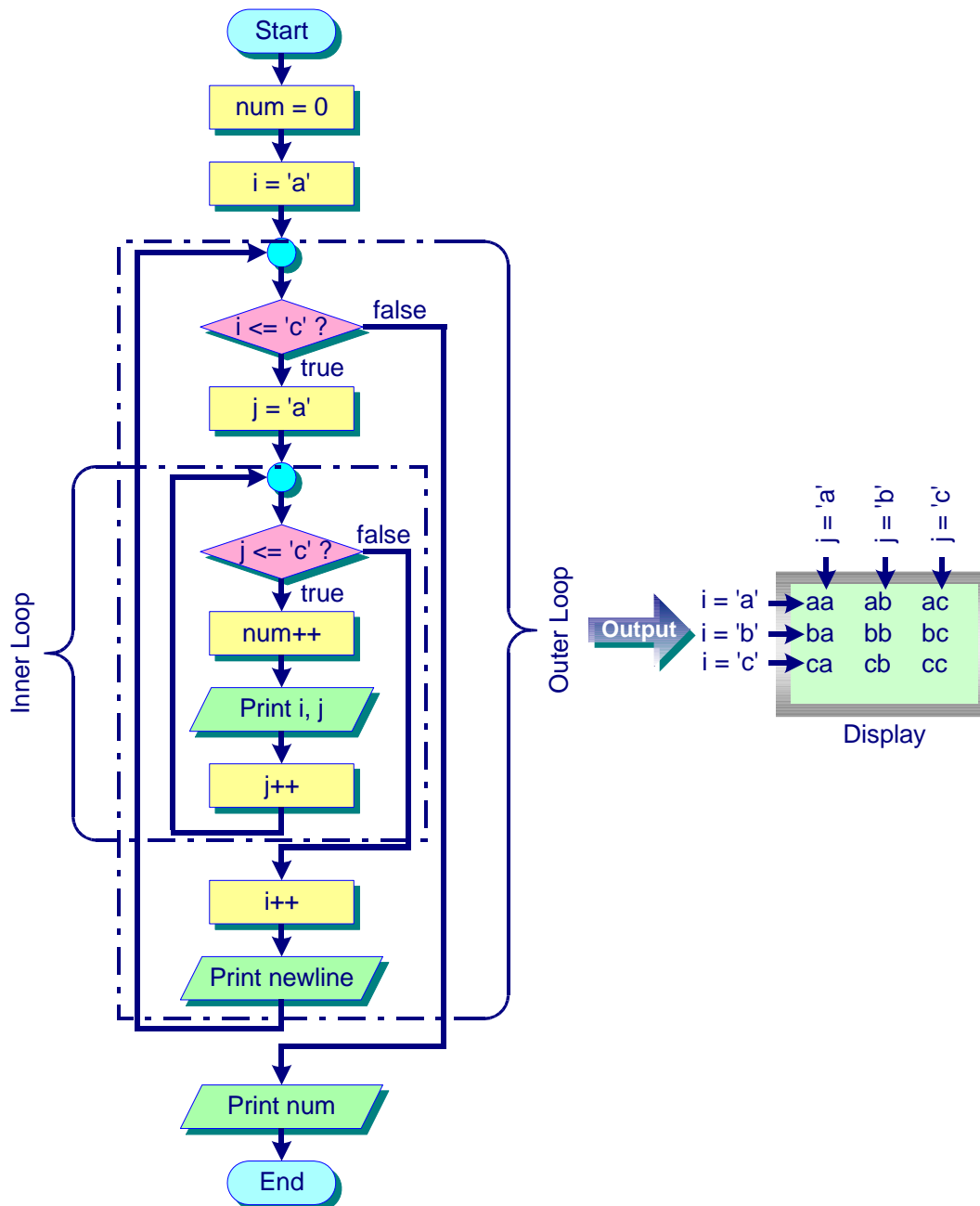


Figure 7.12 Flowchart for Program 7.10.

Another example, which uses nested loop, is given in Program 7.11. The program prints a triangular pattern according to the height entered by the user. For example, when the user enters 5, the pattern with 5 lines is shown in the program output. In this program, a nested **for** loop is used. The outer **for** loop is used to control the line number (i.e. the vertical direction). The loop control variable **lines** is used for this purpose. For each line to be printed, another two inner **for** loops are created. The first inner **for** loop is used to control how many spaces are to be printed, and the second inner **for** loop is used to control how many asterisks '*' are to be printed. The first inner **for** loop will use the relationship between the height of the pattern and the line number to determine the number of spaces to be printed. The number of spaces to be printed is (**height - lines**) for each line. The second inner **for** loop uses the line number, i.e. (**2*lines - 1**) to determine how many asterisk character '*' is to be printed. The pattern is then printed according to the user input on the **height** of the pattern.

Program 7.11 Printing a pattern using nested for loop.

```
#include <stdio.h>

main(void)
{
    int a, b, height, lines;

    printf("Please enter the height of the pattern: ");
    scanf("%d", &height);

    for (lines=1; lines<=height; lines++) {
        for (a=1; a<=(height - lines); a++)
            putchar(' ');
        for (b=1; b<=(2*lines - 1 ); b++)
            putchar('*');
        putchar('\n');
    }
    return 0;
}
```

Program input and output

```
Please enter the height of the pattern: 5
*
***
*****
*****
*****
```

7.8 Exercises

1. Write a program to read in 10 integers and print out the total and the smallest value among them.
2. Write a program that asks the user to input a number, reads the number from the user and prints a triangular pattern. The number to be printed and the height of the triangle are the same number entered by the user. Assume that the user will key in a number between 1 and 9, and no error checking on input is required. For example, when the user keys in 7, the following pattern is produced:

```

      7
     77
    777
   7777
  77777
 777777
7777777

```

3. Write a C program that reads a number from a user and prints a diamond pattern. For example, when the user enters 7, the following pattern is shown. Use only the function `putchar()` to print the pattern.

```

      *
     ***
    *****
   *********
  *******
 *****
  ***
   *

```

4. Write a C program that reads x , as a double-typed number, and prints the result of the following formula.

$$x - \frac{1}{2} \times \frac{x^3}{3} + \frac{1 \times 3}{2 \times 4} \times \frac{x^5}{5} - \frac{1 \times 3 \times 5}{2 \times 4 \times 6} \times \frac{x^7}{7} + \dots - \frac{1 \times 3 \times \dots \times 17}{2 \times 4 \times \dots \times 18} \times \frac{x^{19}}{19}$$

Assume that $-1 \leq x \leq 1$. The program does not need to do error checking.

5. Explain the purpose of the `break` and `continue` statements. Show the output of the following program:

```
#include <stdio.h>
#define MAX 10
main( )
{
    int i=0,j=10;
    do {
        switch(i) {
            case 4: j++;
            case 0: break;
            case 6: i++; continue;
            case 8: j++; break;
            default: break;
        }
        i+=1;
    } while (i<MAX);
    printf("The values are i = %d and j = %d.\n", i,
        j);
}
```

6. Determine the outputs of the following function. Modify the function so that it does not contain break and continue statements, and produces an output identical to the original function.

```
void g( )
{
    int x = 0, y = 10, total =0;
    while (1) {
        if (x >= y) break;
        x += 1;
        if (x % 2 == 0) continue;
        total += x;
        printf ("x = %d, total = %d\n", x, total);
    }
}
```




Functions

To solve real-world problems, we need larger programs than the ones presented in the previous chapters. When developing large and complex programs, the programming task will be much simplified if we use functions. Every C program is made up of one or more functions. A function is a self-contained unit of code to carry out a specific task. Any C functions can call any of the other functions in a program. One of the functions must be named as **main()**. Program execution begins with **main()** and terminates when the last statement of the **main()** function has been executed. In this chapter, we discuss the concepts of functions.

This chapter covers the following topics:

- 8.1 Function Definition
 - 8.2 Function Prototypes
 - 8.3 Calling a Function
 - 8.4 Call by Value
 - 8.5 Pointers
 - 8.6 Call by Reference
 - 8.7 Recursive Functions
 - 8.8 Functional Decomposition
 - 8.9 Placing Functions in Different Files
-

8.1 Function Definition

A function definition is a piece of code, which specifies the actions of the function, and the local data used by the function. A function definition has the following structure:

```

function_header
{
    function_body
}

```

Each function definition consists of a function header and a function body.

Function Header

The *function header* has the following format:

```
type function_name(parameter_list)
```

where **function_name** is the name given to the function. **type** is the data type of the value returned by the function, it can be **int**, **float**, **char**, **void**, etc.

```

int successor(int num)    /* function header */
{                          /* function body begins here */
    return num + 1;
}                          /* function body ends here */

```

The function **successor()** will return a value of the type **int**.

For return type **void**, the function will not return any value. The function **hello_n_times()**

```

void hello_n_times(int n)
{
    int count;
    for (count = 0; count < n; count++)
        printf("Hello\n");
    /* no return statement here */
}

```

prints a string **"Hello"** to the screen the number of times specified by the parameter **n**, which is defined to be of type **int**.

If nothing is specified for **type** of a function header, i.e. when a function is defined with no type, e.g. **main()**, then the default type **int** is used.

The **parameter_list** can be **void** or a list of declarations for variables called *parameters*:

```
type parameterName[,type parameterName]
```

where **[]** can be repeated zero or more times. These parameters are also called *formal arguments*. The parameters are known only inside the function body. They

receive values from the calling function when the function is called. This provides a mechanism for passing data between functions.

In the following `distance()` function:

```
double distance(double x, double y)
{
    return sqrt(x * x + y * y);
}
```

the function is defined with two parameters, `x` and `y`, of data type `double`. It returns a value of type `double`.

In the following `hello()` function:

```
void hello(void)
{
    printf("hello\n");
}
```

the `parameter_list` is `void`, which means that no data will be passed into this function. In addition, the function will not return any value to the calling function.

Function Body

The function body consists of a group of *statements*, it can be declaration statements, simple statements or compound statements. The statements are executed when the function is called. The variables declared inside the function body are called *local* variables and are only known within the function. An example is given in Program 8.1. The function `fn()` expects two parameters of type `int` and returns a value of type `int`. There are two variables defined in the function. Both variables are local variables and can only be accessed within the function. The variables are created when the function is called, and destroyed when the function ends. The function `expn()` expects one parameter of type `float`. It returns a value of type `float`. A variable is also declared in the function. Although the variable has the same name as the one in the function `fn()`, they are two different variables. This variable is only accessible within the function `expn()`. It is also created when the function is called, and will be destroyed when the function exits.

Program 8.1 Functions with parameters.

```
#include <stdio.h>
....
main(void)
```

```

{
    char reply;      /* two local variables in main() */
    int num;
    ....
}
int fn(int x, int y)
{
    float fnum;      /* two local variables in fn() */
    int temp;
    ....
}
float expn(float n)
{
    float temp;      /* a local variable in expn() */
    ....
}

```

The return Statement

The **return** statement is used for functions that return a value. The syntax for the **return** statement is

```
return (expression);
```

The **return** statement may appear in any place and in more than one place inside the function body. Also, the **return** statement terminates the execution of the function and passes the control back to the calling function. An example is shown in Program 8.2. The function **fact()** has **return** statements in various locations in the function body. If **n** is less than 0, then an error message is displayed, and the control is returned to the calling function. If **n** equals 0, then the function returns a value 1. If **n** is greater than 0, then the factorial of **n** is evaluated using a **for** loop, and the result is returned.

Program 8.2 A function that returns a value.

```

#include <stdio.h>
int fact(int n);

main(void)
{
    int n, num;
    printf("Enter a positive number: ");
    scanf("%d", &n);
    num = fact(n);
    printf("The factorial of %d is %d\n", n, num);
}

```

```

    return 0;
}
int fact(int n)
{
    int temp = 1;

    if (n < 0) {
        printf("Error: negative number is not allowed.\n");
        return 0;
    }
    else if (!(n))
        return 1;
    else
        for (; n > 0; n--)
            temp *= n;
    return temp;
}

```

Program input and output

```

Enter a positive number: 4
The factorial of 4 is 24

```

The value returned by the function can be assigned to a variable or used in other functions. This is shown in Figure 8.1. The **main()** function calls the function **fact()**. The value 24 returned by the function **fact()** is then assigned to the variable **num**.

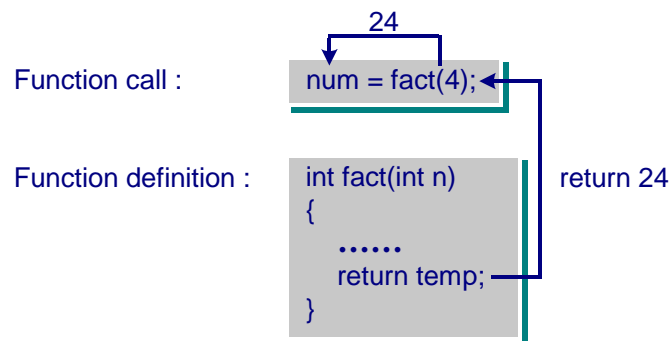


Figure 8.1 Calling the `fact()` function in Program 8.2.

A type **void** function may also have a **return** statement to terminate the function. However, it must not have a **return** expression. An example is given in

Program 8.3. If the function does not have a **return** statement, then the control will be passed back to the calling function when the closing brace of the function body is encountered.

Program 8.3 A void function with a return statement.

```
#include <stdio.h>
void hello_n_times(int n);

main(void)
{
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    hello_n_times(n);
    return 0;
}

void hello_n_times(int n)
{
    int count;
    if (n <= 0)
        return;                /* a return statement */
    else
        for (count = 0; count < n; count++)
            printf("Hello!\n");
}
```

Program input and output

```
Enter a number: 0
Enter a number: 2
Hello!
Hello!
```

Sometimes, it is not necessary to use the value returned by a function. This is illustrated in the use of a number of C library functions such as **printf()** and **scanf()** statements. The **printf()** statement returns a value of type **int** that counts the number of characters printed. The **scanf()** statement also returns the number of items that are successfully read. However, if we do not require this information, we do not need to use the return value returned by these functions.

8.2 Function Prototypes

We need to declare a function before using it in the `main()` function or other functions. A function declaration is called a *function prototype*. It provides the information about the type of the function, the name of the function, and the number and types of the arguments. The declaration is the same as the function header but terminated by a semicolon:

```
void hello_n_times(int n);
```

This declaration specifies that the function `hello_n_times()` expects one argument of type `int` and does not return any value. The function prototype can also be declared without giving the argument name:

```
double distance(double, double);
```

A function must be declared before it is actually called. It can be declared either before the `main()` header, inside the `main()` body or inside any function which uses it. If the function prototype is placed before the `main()` function and at the beginning of the program, it makes the function available to all the functions in the program. This is illustrated in Program 8.4. The function `fact()` is declared outside the `main()` and can be used by all the functions in the program. The function `power()` is declared inside the `main()` function. This makes the function callable only within the `main()` function.

Program 8.4 Function prototypes.

```
#include <stdio.h>
int fact(int n);          /* declared outside the main() */

main(void)
{
    int power(int n);     /* declared inside main() */
    int num1, num2;

    num1 = fact(4);
    num2 = power(3);
    return 0;
}
int fact(int n)           /* function definition */
{
    ....
}
int power(int n)          /* function definition */
```

```
{  
    ....  
}
```

Function prototypes enable the compiler to ensure that functions are being called properly. The compiler will check whether the number of arguments and the type of the arguments of the function call match the parameters used in the function definition. Warning messages will be given if the number of arguments is different. Type casting will also be done if the type of the arguments is mismatch.

8.3 Calling a Function

A function is executed when it is called. A function call has the following format:

```
function_name(argument_list);
```

The function can be called by using the function followed by a list of *arguments*. The statement

```
hello_n_times(4);
```

calls the function `hello_n_times()`. One argument is passed to the function. The function does not return a value.

Function arguments can be constants, variables or expressions. In the function calls:

```
hello_n_times(nooftimes);  
hello_n_times(nooftimes * 4);
```

where the argument `nooftimes` is a variable, and `nooftimes*4` is an expression.

We can also call a function that returns a value:

```
dist = distance(2.0, 4.5);
```

The function call has two arguments, separated by a comma. The function also returns a value. The returned value is assigned to the variable `dist`.

Figure 8.2 illustrates the function call process. The `main()` function starts execution. When the function `fn1()` is called, the program transfers the control to the `fn1()` function which then starts execution. As `fn1()` will return a value back to the calling function, the statements in the function body of `fn1()` will be executed until a `return` statement is encountered. Control is then transferred back

to the **main()** function. The value of the variable **result** will be assigned to the variable **x** in **main()**. The next statement after the function call then starts execution. When the second function **fn2()** is called. The control is then transferred to **fn2()**. The function will execute until the end of the function body. Control will then be transferred back to the **main()** function.

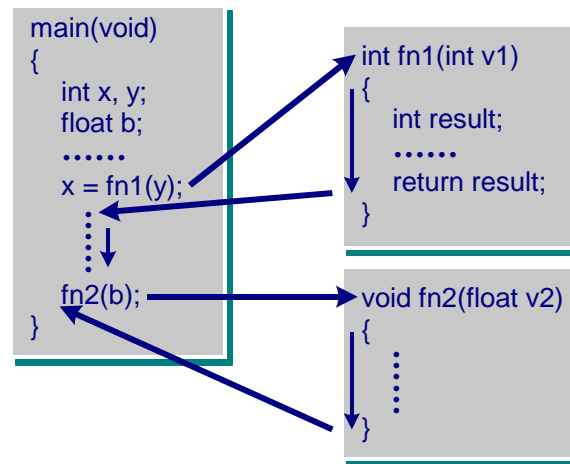


Figure 8.2 Function call process.

Program 8.5 illustrates a program that calls the function **distance()**. When the statement

```
dist = distance(2.0, 4.5);
```

is executed, it calls the function **distance()** in **main()**. Control is then transferred to the function **distance()**. Information is passed between the calling function and the called function through arguments. In this case, the function receives two arguments with values of 2.0 and 4.5. They are assigned to the corresponding formal parameters in the function definition. In addition, we can also use expression as an argument in the function as shown in the following statement:

```
dist = distance(x*y, a*b);
```

When the execution of statements in the function body encounters the **return** statement, the control is then transferred back to the **main()** function, and the statement just after the function call in **main()** will continue to execute.

Program 8.5 Calling a function.

```

#include <stdio.h>
#include <math.h>
double distance (double x, double y);
                                /* function prototype */

main(void)
{
    double dist;
    double x=2.0, y=4.5, a=3.0, b=5.5;
    dist = distance(2.0, 4.5);    /* 2.0, 4.5 - arguments */
    printf("The distance is %lf\n", dist);
    dist = distance(x*y, a*b);    /* x*y, a*b - arguments */
    printf("The distance is %lf\n", dist);
    return 0;
}

double distance(double x, double y) /*  x,y - formal
                                parameters */
{
    return sqrt(x * x + y * y);
}

```

Program output

```

The distance is 4.924429
The distance is 18.794946

```

The names for formal parameters need not be the same as function arguments. However, the number of arguments and the data type of the arguments must be the same as formal parameters defined in function definition. In Program 8.5, the arguments 2.0, 4.5 correspond to formal parameters *x* and *y* in the first function call.

```

dist = distance(2.0, 4.5);          /* function call */
                ↙       ↘
double distance(double x, double y) /* function header */

```

Similarly, the arguments *x*y* and *a*b* in the `main()` function also correspond to formal parameters *x* and *y* in the second function call.

```

dist = distance(x*y, a*b);          /* function call */
                ↙       ↘
double distance(double x, double y) /* function header */

```

8.4 Call by Value

Communications between a called function and the calling function is through *actual arguments*. The called function then performs the task based on the received values. The called function can also return a value back to the calling function. For example, in the function call

```
x = sqrt(y);
```

y is the actual argument that is passed into the function **sqrt()**. The types and the number of actual arguments must match to those of the formal parameters in the function definition.

Before discussing call by value for passing data between functions, it is necessary to distinguish the following three terms: *formal parameters*, *actual parameters* and *arguments*. Formal parameters are used when defining a function. In the function:

```
double distance(double x, double y)
```

x and **y** are formal parameters. They are used to state the type of data to be passed to the function. Actual parameters are used when calling the defined function. In the function call:

```
dist = distance(a, b);
```

where **a** and **b** are actual parameters. The names of the actual parameters can be different from those of the formal parameters. However, they must be of the same type. Arguments are used to state the actual values to be passed to the called function:

```
a = 2.01; b = 4.5;  
dist = distance(a, b);
```

The actual values of 2.01 and 4.5 are the arguments. They are passed to the called function.

Parameter passing between functions may be performed in two ways: *call by value* and *call by reference*. In call by value, the formal parameters must be declared in the function definition as regular variables. The actual parameters in function calls can be constants, variables or expressions. When the function is called, the formal parameters hold a *copy* of the actual parameters. Therefore, changes to the formal parameters in a function are done on the copy of the actual parameters. The actual parameters in the calling function are not affected by anything done in the function.

Program 8.6 illustrates call by value. There are two ways for a called function to return values back to the calling function. The first way is to use the **return** statement as shown in function **add1()**. However, this can only be used when only a single value needs to be passed back from a function. If two or more values need to be passed back from a called function, we need to use another approach called call by reference which will be discussed later in this chapter.

Program 8.6 Call by value.

```
#include <stdio.h>
int add1(int value);
main(void)
{
    int num = 5;
    num = add1(num);
    printf("The value of num is %d\n", num);
    return 0;
}
int add1(int value)
{
    return ++value;
}
```

Program output

The value of num is 6

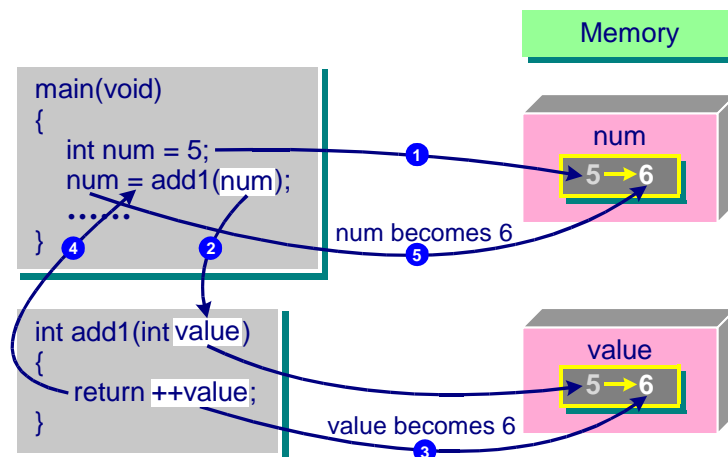


Figure 8.3 Mechanism of call by value illustrated in Program 8.6.

Figure 8.3 shows the mechanism of call by value illustrated in Program 8.6. In Program 8.6, the variable **num** is initially assigned a value of 5 in **main()** (step 1). It is then passed as an argument to the function **add1()** and replaces the formal parameter **value** in the function (step 2). The function **add1()** increments the value stored in the memory location of **value** by 1 (step 3). This value is then returned back to the calling function **main()** (step 4). When received, the value is then assigned to the variable **num** (step 5). The resulting value of **num** is 6.

A function may be called by **main()** or by another function through call by value. An example is shown in Program 8.7. The function **max2()** specifies two formal parameters, **h** and **k**, of type **int**, and receives two function arguments from the calling function. The values of the arguments are then stored in the memory locations of the two parameters, **h** and **k**. The function then compares their values, and returns the larger value back to the calling function.

The function **max3()** specifies three formal parameters, **i**, **j** and **k**, and receives the function arguments from the calling function, and compares their values to determine the largest value. The **max3()** function calls the **max2()** function to compare two values at a time.

```
return max2(max2(i,j), max2(j,k));
```

The function **max2()** is specified in the function **max2()** itself. The returned value from the called function **max2()** will be used again as arguments in the same function **max2()**. The maximum value is then returned back to the calling function.

Program 8.7 Calling other functions in a program.

```
#include <stdio.h>
int max3(int i, int j, int k);
int max2(int h, int k);

main(void)
{
    int x, y, z;

    printf("Enter three integers => ");
    scanf("%d %d %d", &x, &y, &z);
    printf("The maximum value is %d\n", max3(x, y, z));
    return 0;
}
int max3(int i, int j, int k)
{
    return max2(max2(i,j), max2(j,k));
```

```
}  
int max2(int h, int k)  
{  
    printf("Find the max of %d and %d\n", h, k);  
    return h > k ? h : k;  
}
```

Program input and output

```
Enter three integers => 12 24 25  
Find the max of 24 and 25  
Find the max of 12 and 24  
Find the max of 24 and 25  
The maximum value is 25
```

In call by value, the function works on the copy of the actual parameters, the function is prevented from changing the actual parameters. This may prevent any harmful side effects from other functions making changes to a variable that is used in them. As such, it also helps to localize program errors, thereby improving the error debugging process.

8.5 Pointers

Pointer is a very powerful tool for the design of C programs. A pointer is a variable that holds the value of the address or memory location of another data object. In C, pointers can be used in many ways. This includes the passing of variable's address to functions to support call by reference, and the use of pointers for the processing of arrays and strings.

Address Operator

A computer's memory is used to store data objects such as variables and arrays in C. Each memory location has an address that can hold one byte of information. They are organized sequentially and the addresses range from 0 to the maximum size of the memory.

The address of a variable can be obtained by the *address operator* (&). Program 8.8 illustrates the use of address operator. In the statement

```
printf("num = %d, &num = %p\n", num, &num);
```

two values are to be displayed. The first one is the value of 5 that is the initialized value stored at the memory location of the variable `num`. The other is the memory address at which the value of 5 is stored. The address of this memory location is

1024. However, as the memory location is assigned by the computer, it may be different every time the same program is run. We use `&num` to find the value of the address.

The statement

```
scanf("%d", &num);
```

reads in a value of 10 from the standard input, and then stores the value into the address location of the variable `num`. When we perform the `printf()` statement again, the value stored in `num` has been updated to the value of 10. However, the memory address of the variable `num` remains the same throughout the execution of the program.

Program 8.8 Address operator.

```
#include <stdio.h>
main(void)
{
    int num = 5;

    printf("num = %d, &num = %p\n", num, &num);
    scanf("%d", &num);
    printf("num = %d, &num = %p\n", num, &num);
    return 0;
}
```

Program input and output

```
num = 5, &num = 1024
10
num = 10, &num = 1024
```

Pointer Variables

We may have variables, which store the addresses of memory locations of some data objects. These variables are called pointers. A *pointer variable* is declared by:

```
datatype *ptrname;
```

where **datatype** can be any C data type such as `char`, `int`, `float` or `double`. **ptrname** is the name of the pointer variable. The **datatype** is used to indicate the type of data that the variable is pointed to. An asterisk (*) is used to indicate that the variable is a pointer variable. For example, the statement

```
int *ptrI;
```

declares a pointer variable **ptrI** that points to the address of a memory location that is used to store an **int**. The statement

```
float *ptrF;
```

declares a pointer variable **ptrF** that points to the address of a memory location that is used to store a **float**. The statement

```
char *ptrC;
```

declares a pointer variable **ptrC** that points to the address of a memory location that is used to store a **char**.

When a pointer is declared without initialization, memory is allocated to the pointer variable. However, no data or address is stored there.

The value of a pointer is an address. In the statements

```
int a=20; float b=40.0; char c='a';  
int *ptrI; float *ptrF; char *ptrC;  
ptrI=&a; ptrF=&b; ptrC=&c;
```

the variables **a**, **b** and **c**, and pointer variables **ptrI**, **ptrF** and **ptrC** are created. The value of a pointer variable is an address. The pointer variables then point to the memory locations that are used to store the values. The statement

```
ptrI = &a;
```

is used to assign the address of the memory location of **a** to the pointer variable **ptrI**.

Apart from storing an address value in a pointer variable, we can also store the **NULL** value which is defined in `<stdio.h>` in a pointer variable. The pointer is then called a **NULL** pointer. **NULL** is represented in the computer as a series of 0 bits. It refers to the memory location 0. It is common to initialize a pointer to **NULL** in order to avoid it pointing to random memory location.

```
int *ptr = NULL;
```

Figure 8.4 illustrates the pointer variables and the assignment statements.

Indirection Operator

After a pointer variable is assigned to point to a data object or variable, we can access the value stored in the variable using *indirection operator* (*****). If the pointer

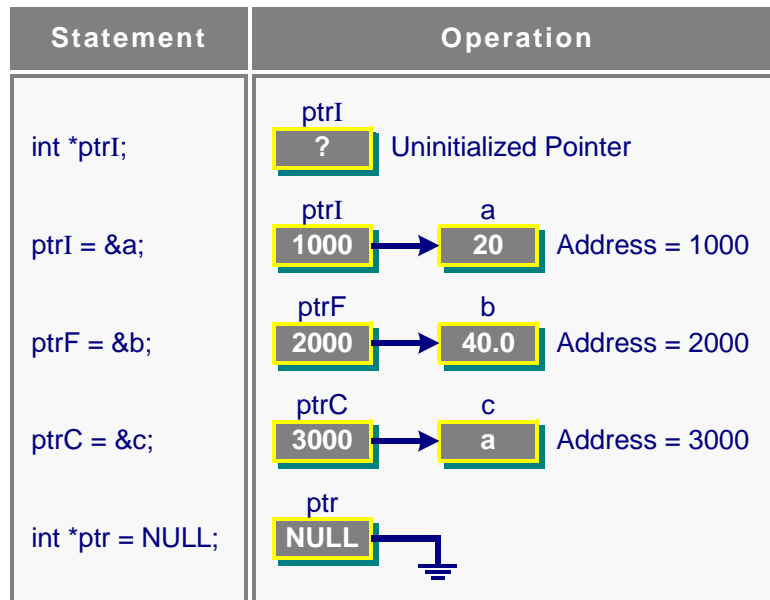


Figure 8.4 Pointer variables.

variable is defined as `ptr`, we use the expression `*ptr` to dereference the pointer to obtain the value stored at the address pointed at by the pointer `ptr`.

Program 8.9 Indirection operator.

```
#include <stdio.h>
main(void)
{
    int num = 3;
    int *ptr;

    ptr = &num;
    printf("num = %d, &num = %p\n", num, &num);
    printf("ptr = %p, *ptr = %d\n", ptr, *ptr);
    *ptr = 10;
    printf("num = %d, &num = %p\n", num, &num);
    return 0;
}
```

Program output

```
num = 3, &num = 1024
ptr = 1024, *ptr = 3
num = 10, &num = 1024
```

Program 8.9 gives a sample program on the use of indirection operator. A pointer variable **ptr** is declared to point to a variable of type **int**. The statement

```
ptr = &num;
```

assigns the address of the variable **num** into the pointer variable **ptr**. The statement

```
printf("ptr = %p, *ptr = %d\n", ptr, *ptr);
```

prints the value of the pointer variable **ptr**, and the content of the memory location pointed to by the pointer variable. This refers to the same value stored at the variable **num**. The statement

```
*ptr = 10;
```

assigns the value 10 to the variable **num**. Since **ptr** points to the address of **num**, the change of value at this memory location has the same effect as changing the value stored at **num**. Therefore, the value stored at **num** is 10. This operation is illustrated in Figure 8.5.

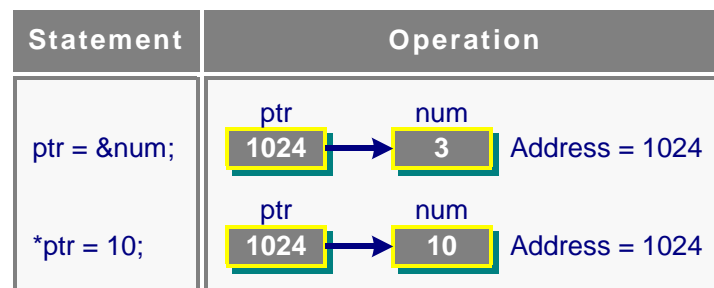


Figure 8.5 Using indirection operator.

Program 8.10 gives another example on using indirection operator. The statements

```
ptr1 = &num1;  
ptr2 = &num2;
```

assign the address of **num1** into **ptr1**, and the address of **num2** into **ptr2**. The statement

```
(*ptr1)++;
```

increments 1 to the content of the memory location pointed to by **ptr1**. The statement

```
*ptr2 = *ptr1;
```

copies the content of the location pointed to by **ptr1** into the location pointed to by **ptr2**. The statement

```
num1 = *ptr2;
```

copies the content of the memory location pointed to by **ptr2** into **num1**. The last statement

```
ptr2 = ptr1;
```

copies the address in **ptr1** into **ptr2**, so that the pointer variable **ptr2** points to the same memory location as **ptr1**.

Program 8.10 Using indirection operator.

```
#include <stdio.h>
main(void)
{
    int num1 = 3, num2 = 5;
    int *ptr1, *ptr2;

    ptr1 = &num1;    /* assign the address of num1 to ptr1 */
    printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);

    (*ptr1)++;        /* increment the content of (*ptr1) */
    printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);

    ptr2 = &num2;    /* assign the address of num2 to ptr2 */
    printf("num2 = %d, *ptr2 = %d\n", num2, *ptr2);

    *ptr2 = *ptr1;    /* copy the content of *ptr1 to *ptr2 */
    printf("num2 = %d, *ptr2 = %d\n", num2, *ptr2);

    *ptr2 = 10;        /* 10 is copied into *ptr2 */
    num1 = *ptr2;    /* copy *ptr2 into num1 */
    printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);

    *ptr1 = *ptr1 * 5;
    printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);
}
```

```

    ptr2 = ptr1;    /* address in ptr1 is copied into ptr2 */
    printf("num2 = %d, *ptr2 = %d\n", num2, *ptr2);
    return 0;
}

```

Program output

```

num1 = 3, *ptr1 = 3
num1 = 4, *ptr1 = 4
num2 = 5, *ptr2 = 5
num2 = 4, *ptr2 = 4
num1 = 10, *ptr1 = 10
num1 = 50, *ptr1 = 50
num2 = 10, *ptr2 = 50

```

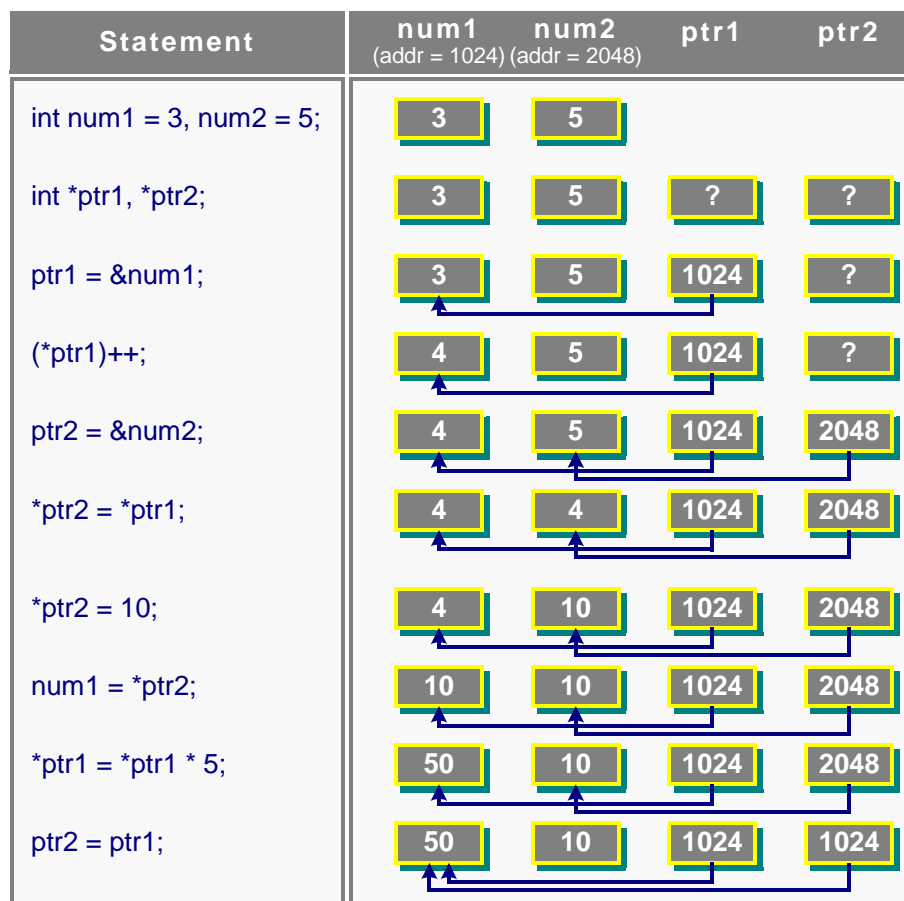


Figure 8.6 Using pointer variables and indirection operators.

Figure 8.6 illustrates the concept of using the pointers given in Program 8.10. In summary, the following illustrates the concepts in using pointer variables.

- **ptr1, ptr2** - They refer to the values stored in the pointer variables **ptr1** and **ptr2**.
- **&ptr1, &ptr2** - They refer to the memory addresses of the variables **ptr1** and **ptr2**.
- ***ptr1, *ptr2** - They refer to the values whose memory locations are stored in the memory locations of the pointer variables **ptr1** and **ptr2**.

8.6 Call by Reference

In call by reference, formal parameters hold the addresses of the actual arguments, i.e. a formal parameter is a pointer. Therefore, changes to the values pointed to by the formal parameter change the actual arguments.

The actual parameters must be the addresses of variables that are local to the calling function. In a function call, the actual arguments must be pointers (or using address operator as the prefix). In the formal parameter declaration list, the formal parameters must be prefixed by the redirection operator. Program 8.11 shows an example program that uses call by reference. Figure 8.7 illustrates the concept of call by reference using Program 8.11 as an example.

Program 8.11 Call by reference.

```
#include <stdio.h>
void add2(int *ptr);

main(void)
{
    int num = 5;
    add2(&num);
    printf("Value of num is %d\n", num);
    return 0;
}

void add2(int *ptr)
{
    ++(*ptr);
}
```

Program output

```
Value of num is 6
```

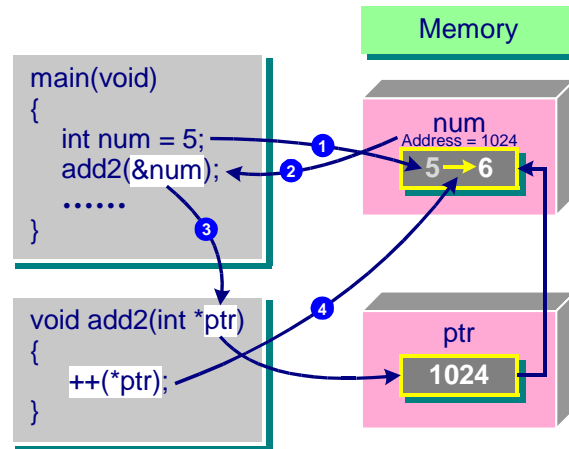


Figure 8.7 Concept of call by reference.

In Program 8.11, the variable `num` is initially assigned a value of 5 in `main()`. The address of the variable `num` is then passed as an argument to the function `add2()` (step 2) and replaces the formal parameter `ptr` in the function (step 3). In `add2()`, the value of the memory location pointed to by the variable `ptr` (i.e. `num`) is then incremented by 1 (step 4). That is, the value stored in the variable `num` becomes 6. When the function ends, the control is then returned to the calling `main()` function.

Program 8.12 is used to further illustrate the concept on call by value and call by reference.

Program 8.12 Call by value and call by reference.

```
#include <stdio.h>
void function1(int a, int *b);
void function2(int c, int *d);
void function3(int h, int *k);

main(void)
{
    int x, y;
    x = 5; y = 5;
    /* location (i) */
    printf("x = %d, y = %d\n", x, y);
    function1(x, &y);
    /* location (x) */
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

```
void function1(int a, int *b)    /* a - call by value, */
                                /* b - call by reference */
{
    /* location (ii) */
    printf("a = %d, *b = %d\n", a, *b);
    *b = *b + a;
    /* location (iii) */
    printf("a = %d, *b = %d\n", a, *b);
    function2(a, b);
    /* location (ix) */
    printf("a = %d, *b = %d\n", a, *b);
}

void function2(int c, int *d)
{
    /* location (iv) */
    printf("c = %d, *d = %d\n", c, *d);
    *d = *d * c;
    /* location (v) */
    printf("c = %d, *d = %d\n", c, *d);
    function3(c, d);
    /* location (viii) */
    printf("c = %d, *d = %d\n", c, *d);
}

void function3(int h, int *k)
{
    /* location (vi) */
    printf("h = %d, *k = %d\n", h, *k);
    *k = *k - h;
    /* location (vii) */
    printf("h = %d, *k = %d\n", h, *k);
}
```

Program output

```
x = 5, y = 5
a = 5, *b = 5
a = 5, *b = 10
c = 5, *d = 10
c = 5, *d = 50
h = 5, *k = 50
h = 5, *k = 45
c = 5, *d = 45
a = 5, *b = 45
x = 5, y = 45
```

The corresponding outputs of the variables and parameters at locations (i) - (x) are given in Table 8.1. The unfilled entries (-) have undefined values.

Table 8.1 Results of Program 8.12.

Location	x	y	a	*b	c	*d	h	*k
(i)	5	5	-	-	-	-	-	-
(ii)	5	5	5	5	-	-	-	-
(iii)	5	10	5	10	-	-	-	-
(iv)	5	10	5	10	5	10	-	-
(v)	5	50	5	50	5	50	-	-
(vi)	5	50	5	50	5	50	5	50
(vii)	5	45	5	45	5	45	5	45
(viii)	5	45	5	45	5	45	-	-
(ix)	5	45	5	45	-	-	-	-
(x)	5	45	-	-	-	-	-	-

Figure 8.8 illustrates the concept shown in Program 8.12. In the function definitions:

```
void function1(int a, int *b)
void function2(int c, int *d)
void function3(int h, int *k)
```

the parameters **a**, **c** and **h** are passed into the functions using call by value, whereas the parameters **b**, **d** and **k** are passed into the functions using call by reference, i.e. addresses are passed to the functions instead of actual values. In fact, the memory contents of these parameters contain the address of the variable **y** in the **main()** function. Any changes to the dereferenced pointers such as ***b**, ***d** and ***k** refer indirectly to changes to the contents stored in the memory location of the variable **y**.

Generally, call by reference is used when we need to pass more than one value back from a function, or in the case that when we use call by value, it will result in a large piece of information being copied to the formal parameter. This could happen when we pass a large array size or structure record.

8.7 Recursive Functions

From the previous sections, we can summarize a few points about functions:

- A function, when called, will accomplish a certain job.

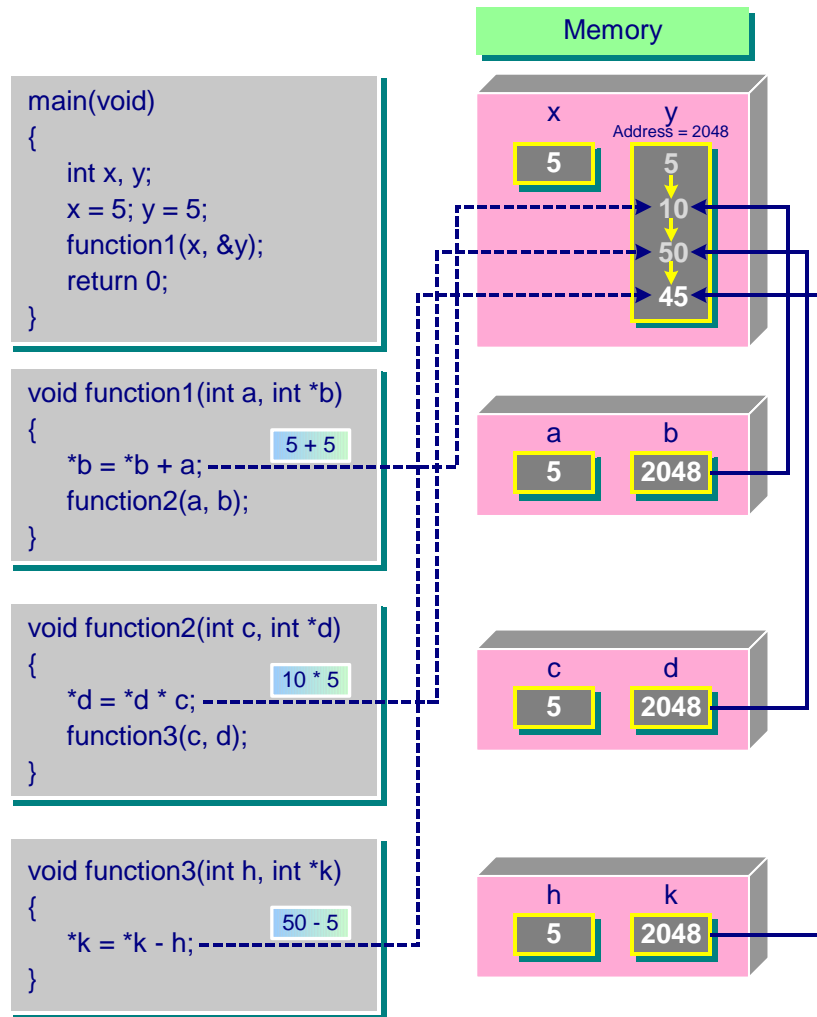


Figure 8.8 Call by reference in Program 8.12.

- When a function $F()$ is called, control is transferred from the calling point to the first statement in $F()$. After the function finishes, the control will return back to the calling point. The next statement after the function call will be executed.
- Each call to a function has its own set of values for the formal arguments and local variables.

These points are important for the understanding of *recursive functions*. A recursive function is a function that calls itself. When a function calls itself, the execution of the current function is suspended, the information that it needs to continue execution is saved, and the recursive call is then evaluated. When the recursive call is evaluated, a new set of variables is created, which is independent from the variables that are created from the previous calls to the function. When

the recursive call is completed, the evaluation result is passed back to the previous call, until the process is completed.

Consider the problem of finding the factorial of a number n :

$$n! = n * (n-1) * (n-2) * \dots * 1$$

The program shown in Program 8.13 implements the factorial function using a **for** loop iteratively. The function **fact1()** receives the actual argument from the calling function. In the **for** loop, the factorial of the input argument is calculated. The result is then passed back to the calling function.

Program 8.13 A factorial function using a for loop.

```
#include <stdio.h>
int fact1(int n);
main(void)
{
    int num;
    printf("Enter a positive integer => ");
    scanf("%d", &num);
    printf("n! = %d\n", fact1(num));
    return 0;
}
int fact1(int n)
{
    int i;
    int temp = 1;
    for (i = n; i > 0; i--)
        temp *= i;
    return temp;
}
```

Program input and output

```
Enter a positive integer => 4
n! = 24
```

Mathematically, a recursive definition of the factorial function can also be defined as:

$$\begin{aligned} 0! &= 1 \\ n! &= n(n-1)! \end{aligned}$$

That is, $n!$ is defined in terms of $(n-1)!$. Based on this definition, we can write

the recursive function `fact2()` as shown in Program 8.14.

Program 8.14 A recursive factorial function.

```
int fact2(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fact2(n-1);
}
```

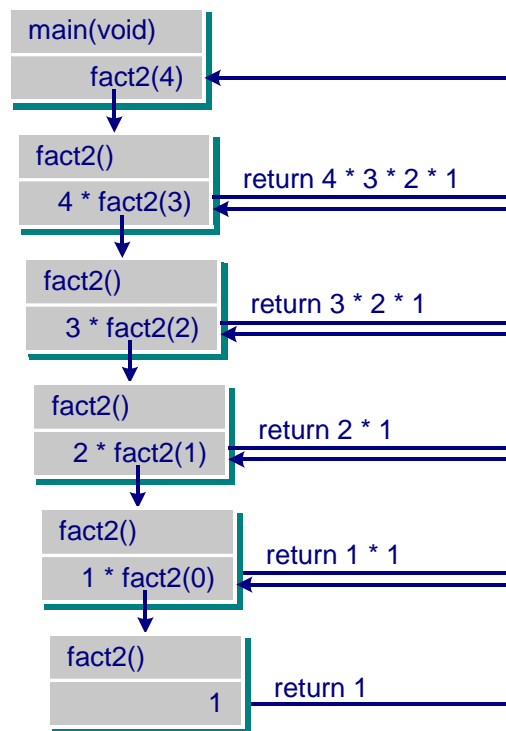


Figure 8.9 Recursive function calls.

To evaluate the function `fact2()` shown in Program 8.14, assume that the user enters 4 for the input value, the result should be 4 factorial ($4!$) which equals to $4 * 3 * 2 * 1$, or 24. The `fact2()` function is recursive, as it calls itself in the `return` statement:

```
return n * fact2(n-1);
```

There are two **return** statements in the function. The first **return** statement

```
return 1;
```

does not call the function **fact2()**. This is the terminating condition, where the recursion will stop. For each new call to **fact2()**, the value **n** gets smaller by 1, and eventually equals to 0. When **n** equals 0, i.e. **0!**, the terminating condition is reached, and the function returns the value of 1. Figure 8.9 shows the values associated with the **n** parameter and the **return** values for each level of recursion.

Notice that each level of the recursive function has its own set of variables. For each level of recursion, a recursive call invokes **fact2()**. When each recursive **fact2()** function terminates, it returns a value to the previous level (from which it was called). When the last **fact2()** function is called, it returns the value 1, as **0!** equals to 1.

Consider another example that performs division using subtraction operation. Assume that **num1** and **num2** are positive integers, we can set up the recursive function as follows:

```
num1 / num2 = 0                                if num1 < num2
num1 / num2 = 1 + (num1-num2)/num2             if num1 >= num2
```

Program 8.15 shows an example program that implements a recursive division function using call by value. In the function **divide1()**, it receives two arguments, **num1** and **num2** from the calling function. The function calls itself with the **return** statement:

```
return 1 + divide1(num1-num2, num2);
```

For each new function call, a new set of parameters is created and used. The value **num1** also gets smaller, and eventually less than **num2**, where the terminating condition is reached. Then, the function returns a value **0** as follows:

```
return 0;
```

Similar to Program 8.14, the decision on whether the terminating condition or a recursive condition is reached is controlled by an **if** statement. The function returns the result to the calling function.

Program 8.15 Recursive division function using call by value.

```
#include <stdio.h>
int divide1(int num1, int num2);
main(void)
```

```

{
    int quotient;
    int x, y;

    printf("Enter two numbers: ");
    scanf("%d %d", &x, &y);
    quotient = divide1(x, y);
    printf("%d / %d = %d\n", x, y, quotient);
    return 0;
}
int divide1(int num1, int num2)
{
    if (num1 < num2)
        return 0;
    else
        return 1 + divide1(num1-num2, num2);
}

```

Program input and output

```

Enter two numbers: 34 6
34 / 6 = 5

```

Program 8.16 implements the recursive division function **divide2()** using call by reference. This function differs from the previous function in that in **divide2()**, the function returns the result through call by reference instead of returning the result to the calling function directly through the **return** statement. This is achieved through the use of the pointer parameter **res** of type **int**. The other two parameters, **num1** and **num2**, are specified in the function using call by value for passing in data. With this, any modification of the value of ***res** such as

```
*res = *res + 1;
```

is referred to the content of the memory location of the variable **quotient** defined in the **main()** function. The terminating condition and the recursive call are the same as Program 8.15.

Program 8.16 Recursive division function using call by reference.

```

#include <stdio.h>
void divide2(int num1, int num2, int *res);
main(void)
{
    int quotient;

```

```
int x, y;

printf("Enter two numbers: ");
scanf("%d %d", &x, &y);
divide2(x, y, &quot;quot;);
printf("%d / %d = %d\n", x, y, quotient);
return 0;
}
void divide2(int num1, int num2, int *res)
{
    if (num1 < num2)
        *res = 0;
    else {
        divide2(num1-num2, num2, res);
        *res = *res + 1;
    }
}
```

Program input and output

Enter two numbers: 34 6
34 / 6 = 5

From the above examples, we have observed the following properties of recursive functions:

- Each function has a terminating condition where no more call to it will be made.
- Each function makes a call to itself with an argument, which is closer to the terminating condition.
- Each level of the function call has its own arguments.
- When a recursive call is made, control is transferred from the calling point to the first statement of the recursive function. When a call at a certain level is finished, control returns to the calling point one level up.

How can one decide on whether to use iteration or recursion approach for implementing functions? The main advantage of using recursive functions is that when the problem is recursive in nature, a recursive function results in shorter and clearer code. However, the main disadvantage of using recursive functions is that recursion is more expensive than iteration in terms of memory usage. Any problems that can be solved recursively can also be solved iteratively (by using loops). A recursive approach is normally chosen over an iterative approach when the recursive approach can more naturally mirror the problem which results in a program that is easier to understand and debug.

8.8 Functional Decomposition

Functional decomposition basically means the top-down stepwise refinement technique discussed in Chapter 2. It starts with the high level description of the program and decomposes the program into successively smaller components until we arrive at a set of suitably sized functions. Then, we design the code for the individual functions using stepwise refinement. At each level of refinement, we are only concerned with what the lower level functions will do. Functional decomposition produces smaller functions that are easier to understand. Smaller functions promote software reusability. However, if functions are very small, we need many of them. In general, a function should be no longer than a page. It will even be better if a function is no longer than half a page.

Using the functional decomposition and top-down stepwise refinement technique, a problem is broken up into a number of smaller subproblems or functions. We then develop the algorithms for the functions. These functions can then be implemented using a programming language. These functions are also called *modules*. This approach of designing programs as functional modules is called *modular design*. The functions or modules should be small and self-contained, so that they can be developed and tested separately. They should also be independent of each other. There are a number of advantages for modular design. Modular programs are easier to write and debug, since they can be developed and tested separately. Another advantage is that modular programs can be developed by different programmers as each programmer can work on a single module of the program independently. Moreover, a library of modules can be developed which can then be reused in other programs that require the same implementation. This can reduce program development time and enhances program reliability. Therefore, modular design can simplify program development significantly.

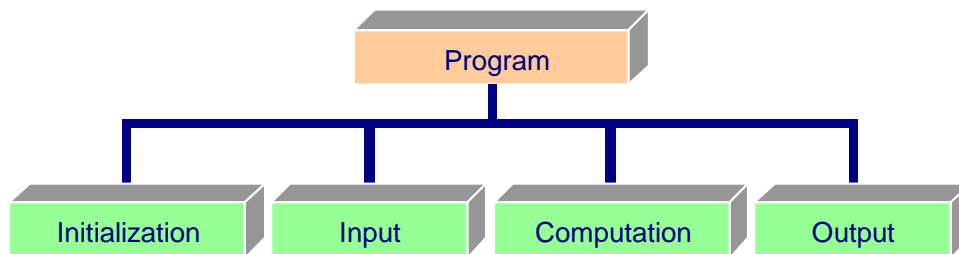


Figure 8.10 A typical structure of a program and its functions.

Figure 8.10 shows a typical structure of a program and its functions for solving a problem. The major functions for a program consists of *Initialization*, *Input*,

Computation and Output. Usually, the functions can be quite complex as well, and they can be divided further into smaller functions. For example, *Computation* may comprise more than one functions.

8.9 Placing Functions in Different Files

A library contains some general-purpose functions that we expect to use in the future. We have used standard libraries to help develop structured programs. Therefore, if we can develop functions in libraries and implement them, we may use them in many other programs, thereby improving software reusability. To achieve this, we need to organize our programs. Each program should consist of one or more header files, one or more implementation files, and the application file.

Header and implementation files are C source files. A header file will have only declarations of functions, global variables, named constants, typedefs and enumeration data types. It also contains the function prototypes of the implemented functions. The extension for the header file is **.h**, while the implementation file is **.c**. Implementation files contain the implementations of the functions declared in the header file. Application file is also a source code file. It contains the **main()** function and any other related functions that are not implemented in the implementation files. Both the implementation file and header file can contain preprocessor directives for the standard libraries such as **#include <stdio.h>**.

To illustrate the above concept on the organization of program files, we may have a program comprising three files. File 1 is called **mainF.c**, which contains the **main()** function as follows:

```
#include <stdio.h>
#include "def.h" /* located in the current directory */
main(void)
{
    ....
    count = function1(h, k);
    function2(&h, &k);
    ....
}
```

This is the program application file. The **main()** body calls **function1()** and **function2()**. The two supporting functions are stored in file 2 called **support.c**. This file is the implementation file.

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



```

#include "def.h"
int function1(int f, int g)
{
    ....
}
void function2(int *p, float *q)
{
    ....
}

```

File 3 (**def.h**) is the header file which contains the constant definitions and the function declarations for the program.

```

#define CONST1 80
#define CONST2 100

int function1(int, int);
void function2(int *, float *);

```

There are two constants, **CONST1** and **CONST2**, defined by **#define** and used by the program. If we do not use a header file, these lines have to be included in both files 1 and 2.

When we have a program comprising several files, the command to compile the program is

```
$ gcc -ansi mainF.c support.c -o mainF
```

The compiler compiles **mainF.c** and produces **mainF.o**, then it compiles **support.c** and produces **support.o**. The linker will produce the executable file **mainF** after linking the two object files and the object files from the library functions.

After successful compilation, if changes are made to **mainF.c** but not **support.c**, then we only need to recompile the **mainF.c** file as follows:

```
$ gcc -ansi mainF.c support.o -o mainF
```

If changes are made to **support.c** but not **mainF.c**, then we only need to recompile the **support.c** file as follows:

```
$ gcc -ansi mainF.o support.c -o mainF
```

In the last two situations, no re-compilation is needed for the file whose **.o** file is given in the command line.

The advantage of placing parts of a program into different files is that the functions in different files can be used by more than one program, and only the files, which have been changed, need to be re-compiled.

8.10 Exercises

1. Explain the terms call by value and call by reference.
2. What is a recursive function? Give an example to illustrate a recursive expression and its terminating condition. What are the advantages and disadvantages of recursive function? Hence, discuss when should a function be implemented recursively and not iteratively.
3. Write a function that displays at the left margin of the screen a solid rectangle whose size has a specified width and length, and inside filled with the characters as specified. The function prototype is

```
void drawRectangle (int width, int length, char chr);
```

For example, `drawRectangle (4, 6, '*')` will display 4 rows and 6 columns of the specified character '*' as follows:

```
*****
*****
*****
*****
```

4. Show the output of the following program.

```
#include <stdio.h>
void fn(int x, int y, int *z);
main( )
{
    int n1=20, n2=15, n3=0;
    fn(n1, n2, &n3);
    printf("n1 = %d, n2 = %d, n3 = %d\n", n1, n2, n3);
    return 0;
}
void fn(int x, int y, int *z)
{
    if (x > y)
    {
        x = x - 2;
```

```

        y = y - 1;
        *z = x * y;
        fn(x,y,z);
    }
    printf("x = %d, y = %d, *z = %d\n", x, y, *z);
}

```

5. Write a function to count the number of digits for a non-negative integer. For example, 1234 has 4 digits. Write two iterative versions of the function that returns the result. The function prototypes are:

```

(a) int iDigits(int n);
(b) void iDigits(int n, int *nd);
    /* the result is returned back through *nd */

```

Write two recursive versions of the function that returns the result. The function prototypes are:

```

(c) int rDigits(int n);
(d) void rDigits(int n, int *nd);
    /* the result is returned back through *nd */

```

6. The function `digitvalue(n,k)` returns the value of the k^{th} digit ($k > 0$) from the right of a non-negative integer n . For example, `digitvalue(1234567, 3) = 5` and `digitvalue(1234, 8) = 0`. Write an iterative version of the function `digitvalue()`. Write a recursive version of the function and call it `rdigitvalue()`.
7. Write a recursive C function `GroupDigits()` which extracts the digits, that are less than 5 from the non-negative number n , and combines the digits sequentially into a new number. If there is no digit that is less than 5, the new number will be set as -1 . The new number is returned via the pointer variable `group1`. The same process is repeated for the digits ≥ 5 and returns the number via the pointer variable `group2`. For example, if $n=123456$, then `*group1 = 1234`, and `*group2 = 56`; if $n=123$, then `*group1 = 123`, and `*group2 = -1`; and if $n=567$, then `*group1 = -1`, and `*group2 = 567`. The function prototype is given as follows:

```
void GroupDigits(long n, long *group1, long *group2);
```

8. The prototype of the recursive function `gcd()` is given as follows:

```
long gcd (long number1, long number2);
```

The function computes the greatest common divisor. For example, the results of the function call for three different pairs of integers are tabulated below.

number1	number2	value returned from <code>gcd()</code>
4	7	1
4	32	4
4	38	2

Implement the function `gcd()`.



Arrays

In the previous chapters, we have discussed the various data types such as **char**, **int**, **float**, etc. When we define a variable of one of these types, the computer will reserve a memory location for the variable. Only one value is stored for each variable at any one time. However, there are applications which require storing related data items under one variable name. For example, if we have different items with similar nature, such as examination marks for the programming course, we might need to declare different variables such as `mark1`, `mark2`, etc. to represent the mark for each student. This is quite cumbersome if the number of students is very large. Instead, we can declare a single variable called `mark` as an array, and each element of the array can be used to store the mark for each student.

In this chapter, we introduce this important topic on data structures that can be used to organize and store related data items. *Arrays* are data structures used to store related data items of the same data type. In Chapter 11, we will discuss other data structures such as *structure* and *union*, which can store related data items of different data types.

This chapter covers the following topics:

- 9.1 Array Declaration
- 9.2 Initialization of Arrays
- 9.3 Operations on Arrays
- 9.4 Pointers and Arrays
- 9.5 Arrays as Function Arguments
- 9.6 Sorting Arrays
- 9.7 Searching Arrays
- 9.8 Multidimensional Arrays
- 9.9 Multidimensional Arrays and Pointers
- 9.10 Multidimensional Arrays as Function Arguments

9.1 Array Declaration

An array (or one-dimensional array) is a collection of elements of *one* data type. The syntax for an array declaration is as follows:

```
type_specifier array_name[array_size];
```

where **type_specifier** specifies the type of data to be stored in the array, **array_name** is the name given to the array, and **array_size** specifies the number of elements in the array. For example, the declaration

```
char name[12];
```

defines an array of 12 elements, each element of the array stores data of type **char**. The elements are stored sequentially in memory. Each memory location in an array is accessed by a relative address called an *index* (or *subscript*).

Arrays can be declared without initialization:

```
float sales[365]; /* array of 365 floats */  
int states[50];   /* array of 50 integers */  
int *pointers[5]; /* array of 5 pointers to integers */
```

When an array is declared, consecutive memory locations for the number of elements specified in the array are allocated by the compiler for the whole array. The total number of bytes of storage allocated to an array will depend on the size of the array and the type of data items. For example, if a system uses 2 bytes to store an integer, the declaration for the array

```
int h[4];
```

will result in a total of 8 bytes allocated for the array as shown in Figure 9.1. The size of memory required can be calculated using the following equation:

```
total_memory = sizeof(type_specifier)*array_size;
```

where **sizeof** operator gives the size of the specified data type and **array_size** is the total number of elements specified in the array.

An integer constant or constant expression must be used to declare the size of the array. Variables or expressions containing a variable cannot be used for the declaration of the size of the array. The following declarations

```
char name[i];    /* i is a variable */
int states[i*6];
```

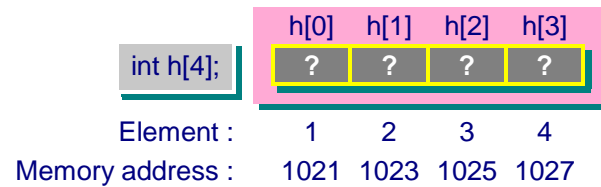


Figure 9.1 An array of 4 elements h[4].

are illegal.

9.2 Initialization of Arrays

After an array has been declared, it must be initialized. Arrays can be initialized at compile time after declaring them. The format for initializing an array is

```
type_specifier array_name[array_size]={list_of_values};
```

The statements

```
#define MTHS 12 /* define a constant */
int days[MTHS]={31,28,31,30,31,30,31,31,30,31,30,31};
```

initialize the array **days** with 12 data items as shown in Figure 9.2.

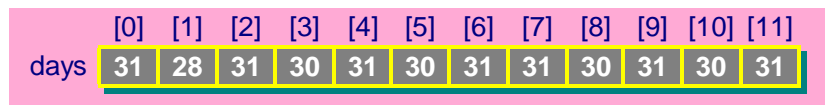


Figure 9.2 Initialization of an array.

An array can also be declared and initialized partially in which the number of elements in the list **{ }** is less than the number of array elements. In the following example, only the first 7 elements of the array are initialized.

```
#define MTHS 12
int days[MTHS]={31,28,31,30,31,30,31};
```

This is illustrated in Figure 9.3. After the first 7 array elements are initialized, the remaining array elements will be initialized to 0.



Figure 9.3 Partial initialization of an array.

In addition, an array can also be declared and initialized without explicitly indicating the array size. The declaration

```
/* an array of 7 elements */
int days[]={31,28,31,30,31,30,31};
```

is valid. It declares **days** as an array of 7 elements as there are 7 elements in the list. This is illustrated in Figure 9.4.

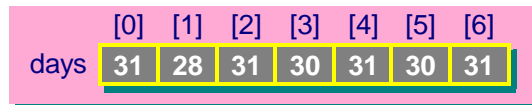


Figure 9.4 Initialization of an array without size specification.

9.3 Operations on Arrays

We can access array elements and perform operations on the array elements. If the array variable **sales** is declared as

```
float sales[365];    /* array of 365 floats */
```

array elements can then be processed. Values can be assigned into each array element. The array can also be used in conditional expressions and looping constructs as follows:

```
sales[0]=143.50;
if (sales[23]==50.0) ...
while (sales[364]!= 0.0) {...}
```


The elements are indexed from 0 to $n-1$ where n is the declared size of the array. Therefore, the following assignment

```
char name[12];
name[12]='c';          /* invalid - index out of range */
```

is invalid since the array elements can only range from `name[0]` to `name[11]`. It is a common mistake to specify an index that is one value more than the largest valid index. Similarly, the following initialization is also invalid.

```
int days[2]={2,3,4,5,6}; /* invalid */
```

To access the address of an array element, we can use the address operator. For example, if we declare an array as

```
int h[5];
```

then `&h[0]` is the address of the 1st element; and `&h[i]` is the address of the $(i+1)^{\text{th}}$ element. The address of an array element is important when performing pointer arithmetic with the array (to be discussed in Section 9.4).

Since array elements can be accessed individually, the most efficient way of manipulating array elements is to use **for** and **while** loops. The loop control variable is used as the index for the array. Thus, each element of the array can be accessed as the value of the loop control variable changes when the loop is executed. This is illustrated in Program 9.1 and Program 9.2. Also note that array values are printed using an index to specify the individual value desired.

In Program 9.1, the array is first initialized using a list. The number in the list should match the size of the array. However, if the list is shorter than the size of the array, then the remaining elements are initialized to 0. If there are too many values in the list compared with the size of the array, this is considered as an error by the compiler. The **for** loop is used as the control construct to print the **days** information.

Program 9.1 Accessing an array using a for loop.

```
#include <stdio.h>
#define MTHS 12          /* define a constant */
main(void)
{
    int i;
    int days[MTHS]={31,28,31,30,31,30,31,31,30,31,30,31};
    /* print the number of days in each month */
    for ( i=0 ; i<MTHS ; i++ )
        printf("Month %d has %d days.\n", i+1, days[i]);
}
```

```

    return 0;
}

```

Program output

```

Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.

```

Program 9.2 finds the maximum non-negative value in an array. Firstly, the value for each item in an array is read from the user and stored in the array. The value `-1` is assigned to the variable `max`, which is defined as the current maximum. Then, the items in the array are checked one by one using a `for` loop. If the value of the next item is larger than the current maximum, it becomes the current maximum. If the value of the next item is less than the current maximum, the current value of `max` is retained. The maximum value in the array is then printed on the screen. Figure 9.5 illustrates the operations of Program 9.2.

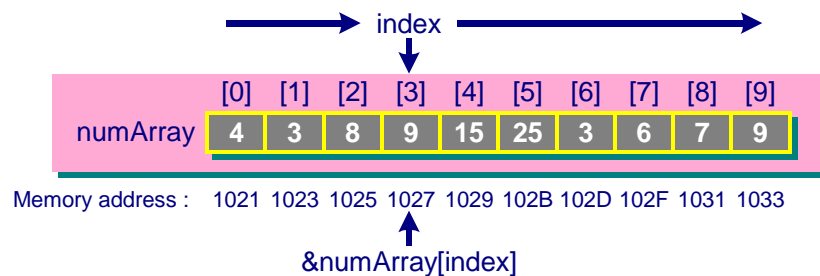


Figure 9.5 Accessing arrays.

Program 9.2 Finding the maximum number stored in an array.

```

#include <stdio.h>

```

```

main(void)
{
    int index, max, numArray[10];

    printf("Enter 10 numbers: \n");
    for (index = 0; index < 10; index++)
        scanf("%d", &numArray[index]);
    max = numArray[0];
    for (index = 1; index < 10; index++)
        if (numArray[index] > max)
            max = numArray[index];
    printf("The maximum value is %d.\n", max);
    return 0;
}

```

Program input and output

```

Enter 10 numbers:
4 3 8 9 15 25 3 6 7 9
The maximum value is 25.

```

9.4 Pointers and Arrays

There is a strong relationship between pointers and arrays. The array name is actually a pointer constant. When the following array is created:

```
int days[12];
```

a *pointer constant* with the same name as the array is also created. The pointer constant points to the first element of the array. Therefore, the array name by itself, **days**, is actually the address (or pointer) of the first element of the array. Assume an integer is represented by 2 bytes and the array **days** begins at memory location 1021. Figure 9.6 illustrates the relationship between the array and the pointer constant. The array consists of 12 elements. The value stored at **days** is 1021, which corresponds to the address of the first element of the array.

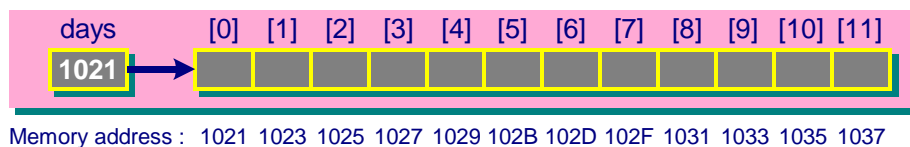


Figure 9.6 Relationship between an array and a pointer.

Since the array name is the pointer to the first element of the array, we have

```
days == &days[0]
*days == days[0]
days + 1 == &days[1]
*(days + 1) == days[1]
```

However, it is important to note that the array name is a pointer constant, not a pointer variable. It means that the value stored in **days** cannot be changed by any statements. The following assignment statements:

```
days += 5;
days++;
```

are invalid.

Program 9.3 Pointer variable and pointer arithmetic.

```
#define MTHS 12
main(void)
{
    int days[MTHS]={31,28,31,30,31,30,31,31,30,31,30,31};
    int *day_ptr;
    day_ptr = days;      /* points to the first element*/
    day_ptr = &days[3]; /* points to the fourth element */
    day_ptr += 3;        /* points to the seventh element */
    day_ptr--;          /* points to the sixth element */
    return 0;
}
```

A *pointer variable* can take on different addresses. An example is illustrated in Program 9.3. **days** is declared as an array of 12 elements, and it is also a pointer constant. **day_ptr** is declared as a pointer variable. The statement

```
day_ptr = days;
```

assigns the value 1021 stored in **days** to the pointer variable **day_ptr**. This causes the pointer variable to also point to the first element of the array. The statement

```
day_ptr = &days[3];
```

assigns the address of `days[3]` to the `day_ptr`. It updates the `day_ptr` to point to the fourth element of the array. The value stored in `day_ptr` becomes 1027. We can also add an integer value of 3 to the pointer variable `day_ptr`.

```
day_ptr += 3;
```

The `day_ptr` will move forward three elements. The `day_ptr` contains the value of 102D, which is the address of the seventh element of the array `days[6]`. The pointer variable can also be decremented as:

```
day_ptr--;
```

The `day_ptr` moves back one element in the array. It points to the sixth element of the array `days[5]`. When we perform pointer arithmetic, it is carried out according to the size of the data object that the pointer refers to. If `day_ptr` is declared as a pointer variable of type `int`, then every two bytes will be added for every increment of one. Figure 9.7 illustrates the pointer arithmetic shown in Program 9.3.

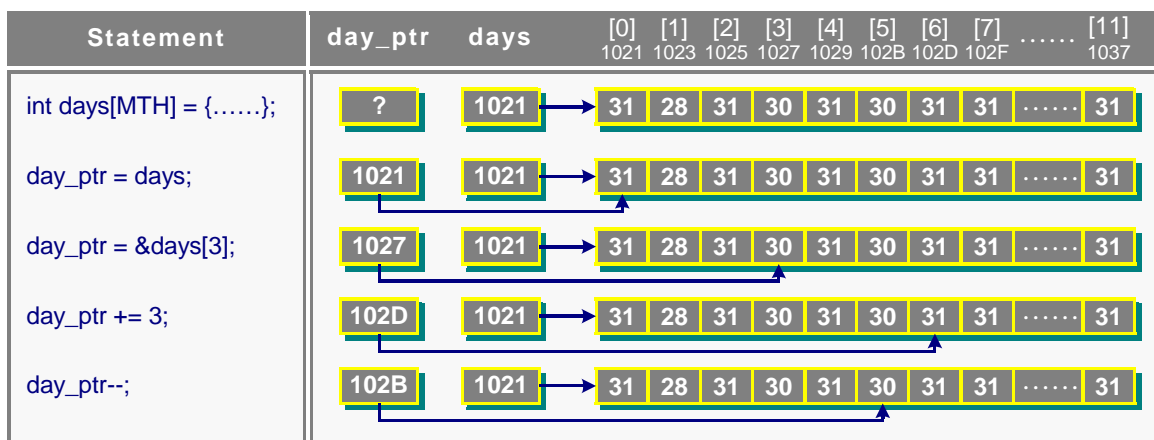


Figure 9.7 Pointer arithmetic in Program 9.3.

Instead of using *array notation*, we can use another approach that uses *pointer notation* to access the individual elements of an array. Program 9.4 produces the same output as Program 9.2 which uses array notation. In Program 9.4, `numArray` is the address of the first element of the array, `numArray+index` is the address of element `numArray[index]`, and `*(numArray+index)` is the value of that element. This is the same as the `numArray[index]`. The loop accesses each element of the array, and compares it with `max` in order to determine the maximum value. The maximum value is then assigned to `max`.

Program 9.4 Finding the maximum number using pointer notation.

```
#include <stdio.h>
main(void)
{
    int index, max, numArray[10];

    printf("Enter 10 numbers: \n");
    for (index = 0; index < 10; index++)
        scanf("%d", numArray + index);
    max = *numArray;
    for (index = 1; index < 10; index++)
        if (*(numArray + index) > max)
            max = *(numArray + index);
    printf("The maximum value is %d.\n", max);
    return 0;
}
```

Program input and output

```
Enter 10 numbers:
6 9 12 5 13 20 8 16 17 21
The maximum value is 21.
```

Program 9.4 uses the pointer constant **numArray** to access all the elements of the array. Another way to access the elements of an array is to use a pointer variable. Program 9.5 gives an example using a pointer variable to find the maximum element of the array. We increment the **ptr** as

```
ptr++;
```

to access each element of the array in order to determine the maximum value stored in the array. Figure 9.8 illustrates the operations shown in Program 9.5.

Program 9.5 Finding the maximum number using a pointer variable.

```
#include <stdio.h>
main(void)
{
    int index, max, numArray[10];
    int *ptr;                      /* pointer variable */

    ptr = numArray;
    printf("Enter 10 numbers: \n");
```

```

    for (index = 0; index < 10; index++)
        scanf("%d", ptr++);
    ptr = numArray;
    max = *ptr;
    for (index = 0; index < 10; index++) {
        if (*ptr > max)
            max = *ptr;
        ptr++;
    }
    printf("The maximum value is %d.\n", max);
    return 0;
}

```

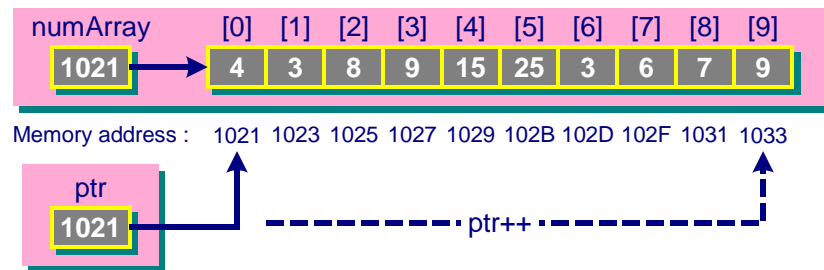


Figure 9.8 Array processing using pointers.

Both the array notation and pointer notation can be used for accessing individual elements of an array. However, the pointer notation will be more efficient than the array notation, and it is also more convenient when working with strings.

9.5 Arrays as Function Arguments

We can use an array in a function's body. We may also use an array as a function argument. An array consists of a number of elements. We may access individual element, or pass an element to a function. To do this, we need to indicate the index of the array, e.g. `days[3]` to specify the specific element that we want to pass to the function.

An array can also be passed to a function as an argument. For example, assume a function called `maximum()` that finds the maximum value of the following array elements:

```
int table[10] = {34,21,65,54,17,48,29,93,49,23};
```

has been written. We may call the function as follows:

```
maximum(table, 10);
```

To receive the array argument, the function must be defined with a formal parameter that is an array. There are three ways to define a function with a one-dimensional array as the argument. The first way is to define the function as

```
int maximum(int table[], int n)
```

The parameter list includes an array **table** and an integer **n**. The data type of the array is specified, and empty square brackets follow the array name. The integer **n** is used to indicate the size of the array. Another way is to define the function as

```
int maximum(int table[TABLESIZE])
```

The parameter list includes an array only. The array size **TABLESIZE** is also specified in the square brackets of the array **table**. The third way is

```
int maximum(int *table, int n)
```

The parameter list includes a pointer **table** of type **int**, and an integer **n**. The integer **n** is used to indicate the size of the array.

The function prototypes of the function become

```
int maximum(int table[], int n);  
or  int maximum(int table[TABLESIZE]);  
or  int maximum(int *table, int n);
```

Program 9.6 shows a sample program that passes an array as the function argument. The function **maximum()** determines the maximum value stored in the array called **table**. The number of elements stored in the array is also passed as an integer argument **n**.

Program 9.6 Passing an array as a function argument.

```
#include <stdio.h>  
int maximum(int table[], int n);  
  
main(void)  
{  
    int max, index, n;  
    int numArray[10];
```



```
printf("Enter the number of values: ");
scanf("%d", &n);
printf("Enter %d values: ", n);
for (index = 0; index < n; index++)
    scanf("%d", &numArray[index]);
max = maximum(numArray, n);
printf("The maximum value is %d.\n", max);
return 0 ;
}
int maximum(int table[], int n)
{
    int i, temp;

    temp = table[0];
    for (i = 1; i < n; i++)
        if (table[i] > temp)
            temp = table[i];
    return temp;
}
```

Program input and output

```
Enter the number of values: 5
Enter 5 values: 12 5 13 20 8 16
The maximum value is 20.
```

Figure 9.9 illustrates the concept on passing an array as a function argument given in Program 9.6. As illustrated in Figure 9.9 and Program 9.6, when we pass an array as a function argument, the original array is passed to the function. There is no local copy of the array to be maintained in the function. This is mainly due to efficiency as arrays can be quite large and thereby taking a considerably large storage space if a local copy is stored. In fact, the array is passed using *call by reference* to the function. This means that the address of the first element of the array is passed to the function. Since the function has the address of the array, any changes to the array are made to the original array.

9.6 Sorting Arrays

Sorting processes and arranges data according to a particular order. A list of numbers or a list of names in alphabetical order can be sorted using sorting algorithms. Different sorting algorithms, such as selection sort, bubble sort, quick sort and merge sort, are available. The efficiency of a sorting algorithm is based on

the number of comparisons it takes during sorting. In this section, we will only introduce the bubble sort. It is one of the simplest sorting algorithms, however, it is inefficient when compared with other sorting algorithms such as merge sort and quick sort.

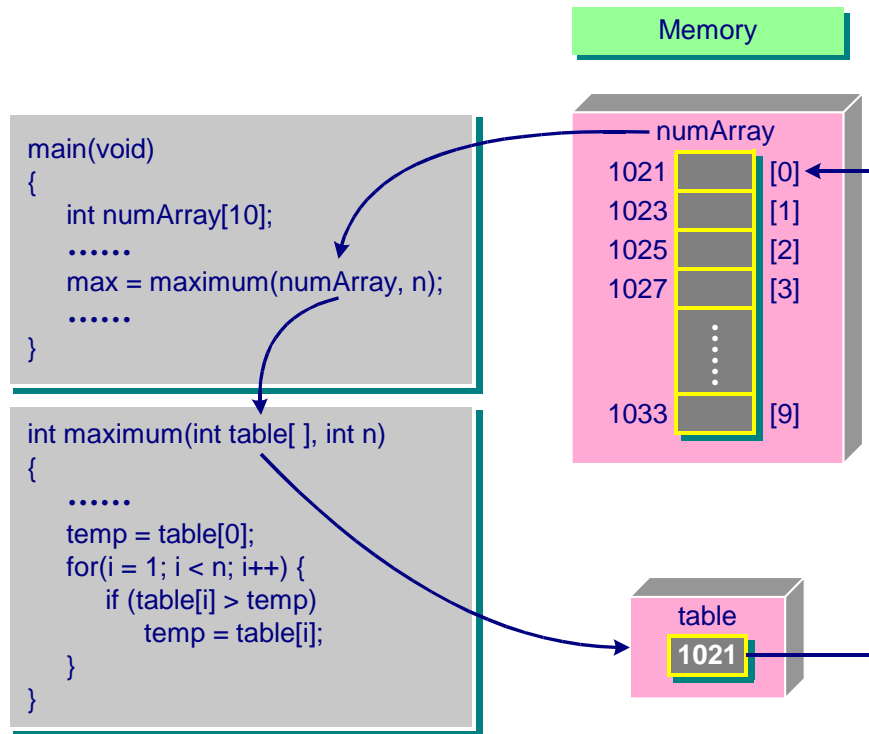


Figure 9.9 Passing an array as a function argument in Program 9.6.

Assuming that we are to use the bubble sort algorithm to sort the following list of five numbers in ascending order:

	[0]	[1]	[2]	[3]	[4]
x	9	2	8	4	1

The bubble sort is operated as follows:

1. Compare the first two numbers, and if the second number is smaller than the first, then the numbers are swapped.
2. Then, move down one number, compare that number and the number that follows it, and swap the two numbers, if necessary.
3. Repeat this sequence until the last two numbers of the array have been compared. Each sequence of comparison is called a pass.

4. The passes stop once all numbers are in the correct order.

To sort the example given above, we need to have several passes. In the first pass, we test the first two numbers at $x[0]$ and $x[1]$. As the second number is smaller than the first, the two numbers are swapped. The list now has the following numbers:

	[0]	[1]	[2]	[3]	[4]
x	2	9	8	4	1

Then, we move down one number to $x[1]$ and compare this number with $x[2]$ that follows $x[1]$. In this comparison, the number at $x[2]$ is smaller than $x[1]$, then the two numbers are swapped. The list now contains the following numbers:

	[0]	[1]	[2]	[3]	[4]
x	2	8	9	4	1

We repeat the process and compare the numbers at $x[2]$ and $x[3]$. Again, the number at $x[3]$ is smaller than $x[2]$, so the two numbers are swapped. The list now looks like the following:

	[0]	[1]	[2]	[3]	[4]
x	2	8	4	9	1

Finally, the last two numbers at $x[3]$ and $x[4]$ are compared and swapped. The list now contains the numbers in the following order:

	[0]	[1]	[2]	[3]	[4]
x	2	8	4	1	9

After the first pass, the largest number has now moved to the bottom of the list. However, the list is not completely in the ascending order yet. We still need to perform the second pass through the list. At the end of the second pass, the list of numbers now looks like the following:

	[0]	[1]	[2]	[3]	[4]
x	2	4	1	8	9

The third pass gives the following list of numbers:

	[0]	[1]	[2]	[3]	[4]
x	2	1	4	8	9

The fourth pass gives the following list of numbers that is already in ascending order:

	[0]	[1]	[2]	[3]	[4]
x	1	2	4	8	9

Therefore, in pass 5, the list of numbers stored in the array has already been sorted, and no swapping is needed during pass 5.

Program 9.7 gives the bubble sort algorithm, which sorts a list of numbers in ascending order. In the bubble sort algorithm, the list of numbers to be sorted is stored in the array `x[]`. The parameter `n` is used to indicate the total number of elements to be sorted. In the `main()` function, it reads in the number of elements, and the corresponding values from the user. The values are stored in the array variable called `number[]`. The sorted list is passed back to the calling function `main()` using the array pointer `number`. In the bubble sort algorithm, two `for` loops are used. The outer loop executes a total of `n-1` passes. The inner loop compares any two numbers for each pass, and makes a swap if necessary. The swapping is performed using the following assignment statements:

```
temp_value = x[index];
x[index] = x[index+1];
x[index+1] = temp_value;
```

Program 9.7 Bubble sort algorithm.

```
#include <stdio.h>
void bubble_sort(int x[], int n);

main(void)
{
    int i, n;
    int number[10];    /* array to be sorted */

    printf("Enter the number of items to be sorted: ");
    scanf("%d", &n);
    printf("Enter the list of numbers: ");
    for (i = 0; i < n; i++)
```

```

        scanf("%d", &number[i]);
    bubble_sort(number, n);
    printf("The sorted array is: ");
    for (i = 0; i < n; i++)
        printf("%d ", number[i]);
    return 0;
}
void bubble_sort(int x[], int n)
{
    int temp_value;
    int pass;
    int index;

    for (pass = 0; pass < n-1; pass++) { /* n-1 passes */
        for (index=0; index < n-1; index++) { /* for one pass */
            if (x[index] > x[index+1]) { /* comparison */
                temp_value = x[index]; /* swap process */
                x[index] = x[index+1];
                x[index+1] = temp_value;
            }
        }
    }
}

```

Program input and output

```

Enter the number of items to be sorted: 6
Enter the list of numbers: 6 2 7 9 1 4
The sorted array is: 1 2 4 6 7 9

```

9.7 Searching Arrays

When working with arrays, it may be necessary to search for the presence of a particular element. The element that needs to be found is called a *search key*. There are many searching algorithms available. In this section, we discuss two searching algorithms: the *linear search* and the *binary search*.

The linear search algorithm compares each element of the array with the search key until a match is found or the end of the array is reached. Program 9.8 gives an example of the linear search technique. In fact, Program 9.2 has demonstrated another example of the linear search by comparing each element of the array with the maximum number of the array. On average, the linear search algorithm requires to compare the search key with half of the elements stored in an array. Linear search is sufficient for small arrays. However, it is inefficient for large and

sorted arrays. Therefore, the more efficient binary search technique should be used for large arrays.

Program 9.8 Searching an array using linear search.

```
#include <stdio.h>
#define SIZE 10
int linearSearch(int array[], int key, int n);

main(void)
{
    int i, searchkey, found, numArray[SIZE];

    printf("Enter a list of 10 numbers: ");
    for (i = 0; i < 10; i++)
        scanf("%d", &numArray[i]);
    printf("Enter the integer key to be searched: ");
    scanf("%d", &searchkey);
    found = linearSearch(numArray, searchkey, SIZE);
    if (found != -1)
        printf("The search value has been found at: %d\n",
            found);
    else
        printf("The search value cannot be found.\n");
    return 0;
}

int linearSearch(int array[], int key, int n)
{
    int index;
    for (index = 0; index < n; index++)
        if (array[index] == key)
            return index;
    return -1;
}
```

Program input and output

```
Enter a list of 10 numbers: 5 1 8 9 3 42 7 0 14 21
Enter the integer key to be searched: 9
The search value has been found at: 3
```

The binary search algorithm first locates the middle element in the array and compares it with the search key. If they match, the search key is found, and the index of the array is returned. If not, then we continue the process to search one half of the array. This is done as follows. If the search key is less than the middle

element of the array, then the first half of the array will be searched. Otherwise, the second half of the array will be searched. This process is repeated until the search key is found. This algorithm is much more efficient than linear search for sorted arrays. For an array of 31 elements, it only takes 5 comparisons to search for the search key.

Program 9.9 gives the iterative version of the binary search algorithm. The function `binarySearch()` is implemented with the `while` loop in which the operations inside are executed until the search key is found or the end of the array is reached without success. The operations perform the finding of the mid-point of the array, comparing the value stored at the mid-point array element with the search key, and searching the one half of the array if the search key is not found.

Program 9.9 Searching an array using binary search.

```
#include <stdio.h>
#define SIZE 10
int binarySearch(int array[], int key, int n);

main(void)
{
    int i, searchkey, found, numArray[SIZE];

    printf("Enter a list of 10 numbers: ");
    for (i = 0; i < 10; i++)
        scanf("%d", &numArray[i]);
    printf("Enter the integer key to be searched: ");
    scanf("%d", &searchkey);
    found = binarySearch(numArray, searchkey, SIZE);
    if (found != -1)
        printf("The search value has been found at: %d\n",
            found);
    else
        printf("The search value cannot be found.\n");
    return 0;
}

int binarySearch(int array[], int key, int n)
{
    int middle;           /* mid-point */
    int first = 0;        /* first position in array */
    int last = n-1;       /* last position in array */

    while (first <= last) {
        middle = (first + last)/2;
        if (key == array[middle]) /* search key found */
            return middle;
    }
}
```

```

        else if (key < array[middle])
            last = middle - 1;
        else
            first = middle + 1;
    }
    return -1;          /* search key cannot be found */
}

```

Program input and output

```

Enter a list of 10 numbers: 2 4 6 8 11 17 21 28 35 41
Enter the integer key to be searched: 21
The search value has been found at: 6

```

Searching algorithms can also be implemented using a recursive approach. Program 9.10 gives two examples for recursive implementation of searching algorithms. Both algorithms count the number of times the integer **key** appears in the array, which has **n** integers in it. In the **count1()** function, it works in a linear search manner in that the function recursively divides the sub-array until the sub-array contains only one element. Then, the element contained in the sub-array is matched with the integer **key**. If they match, the total count is incremented by 1. This repeats recursively for all the sub-arrays, and the total count that indicates the times the integer **key** occurs, can be calculated. In the **count2()** function, it works in a binary search manner. The array is sub-divided into two halves, and it repeats the process recursively until the sub-array contains only one element. The element is then matched with the integer **key**. If they match, then the total count is incremented by 1. The function recursively checks the content for each element of the array, and the total count is calculated.

Program 9.10 Recursive searching algorithms.

```

#include <stdio.h>
#define SIZE 10
int count1(int array[], int key, int n);
int count2(int array[], int key, int n);

main(void)
{
    int i, searchkey, numArray[SIZE];
    int num1, num2;

    printf("Enter a list of 10 numbers: ");
    for (i = 0; i < 10; i++)
        scanf("%d", &numArray[i]);
}

```



```

printf("Enter the integer key to be found: ");
scanf("%d", &searchkey);
num1 = count1(numArray, searchkey, SIZE);
printf("The number of the search value in count1() is
      %d\n", num1);
num2 = count2(numArray, searchkey, SIZE);
printf("The number of the search value in count2() is
      %d\n", num2);
return 0;
}
int count1(int array[], int key, int n)
{
    if (n == 1)
        if (array[0] == key)
            return 1;
        else return 0;
    if (array[0] == key)
        return 1 + count1(&array[1], key, n-1);
    else
        return count1(&array[1], key, n-1);
}
int count2(int array[], int key, int n)
{
    if (n == 1)
        if (array[0] == key)
            return 1;
        else return 0;
    return (count2(&array[0], key, n/2) + count2(&array[n/2],
        key, n-n/2));
}

```

Program input and output

```

Enter a list of 10 numbers: 1 3 5 7 9 7 5 3 1 7
Enter the integer key to be searched: 7
The number of the search value in count1() is 3
The number of the search value in count2() is 3

```

9.8 Multidimensional Arrays

In the previous sections, we have discussed one-dimensional arrays in which only a single index (or subscript) is needed to access a specific element of the array. The number of indexes that are used to access a specific element in an array is called the *dimension* of the array. Arrays that have more than one dimension are

called multidimensional arrays. In the subsequent discussion, we focus mainly on two-dimensional arrays. We may use two-dimensional arrays to represent data stored in tabular form. Two-dimensional arrays are especially useful in matrix manipulation.

A two-dimensional array can be declared as

```
int x[3][5];    /* 3-element array of 5-element arrays */
```

Two indexes are needed to access each element of the array. Similarly, a three-dimensional array can be declared as

```
char x[3][4][5];
```

Three indexes are used to access a specific element of the array. ANSI standard supports arrays with a maximum of 6 dimensions.

The statement

```
int x[3][5];
```

declares a two-dimensional array **x**[] of type **int** having three rows and five columns. The compiler will set aside the memory for storing the elements of the array. The two-dimensional array can also be viewed as a table made up of rows and columns. For example, the array **x**[3][5] can be represented as a table as follows:

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	x[0][0]	x[0][1]	x[0][2]	x[0][3]	x[0][4]
Row 1	x[1][0]	x[1][1]	x[1][2]	x[1][3]	x[1][4]
Row 2	x[2][0]	x[2][1]	x[2][2]	x[2][3]	x[2][4]

The array consists of three rows and five columns. The array name and two indexes are used to represent each individual element of the array. The first index is used for the row, and the second index is used for column ordering. **x**[0][0] represents the first row and first column, and **x**[1][0] represents the second row and first column, and **x**[1][3] represents second row and fourth column, etc. A two-dimensional array is stored in *row-major* order in the memory. Figure 9.10 illustrates the memory storage of the two-dimensional array **x**[3][5].

For the initialization of multidimensional arrays, each row of data is enclosed in braces:

```
int x[2][2] = { {1,2},          /* first row */
               {6,7} };        /* second row */
```

The data in the first interior set of braces is assigned to the first row of the array, the data in the second interior set goes to the second row, etc. If the size of the list in the first row is less than the array size of the first row, then the remaining elements of the row are initialized to zero. If there are too many data, then it is an error.

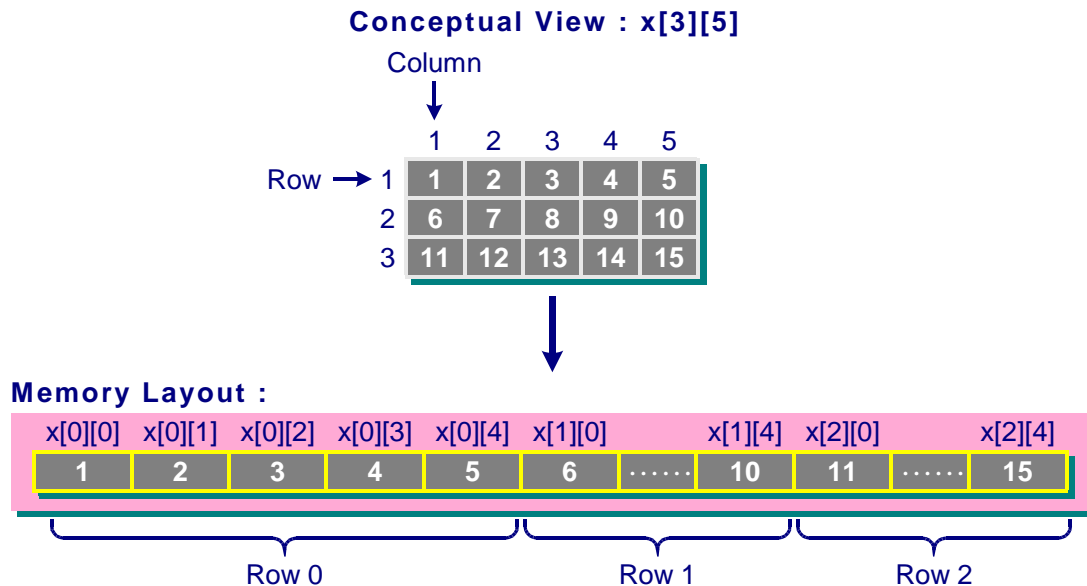


Figure 9.10 Storage of a two-dimensional array in memory.

Since the inner braces are optional, a multidimensional array can be initialized as

```
int x[2][2] = { 1,2,6,7 };
```

An array can also be initialized partially:

```
int exam[3][3] = { {1,2}, {4}, {5,7} };
```

This statement initializes the first two elements in the first row, the first element in the second row, and the first two elements in the third row. All elements that are not initialized are set to zero by default.

The following statement

```
int exam[3][3] = { 1,2,4,5,7 };
```

is valid, but the two-dimensional array is initialized as

```
int exam[3][3] = { {1,2,4}, {5,7} };
```

If the number of rows and columns are not specified in the declaration statement, the compiler will determine the size of the array based on information about the initialization of the array. For example, the following statement

```
int array[][] = { {1,2},
                  {6,7} };
```

will create an array having two rows and two columns. The array is also initialized according to the values specified in the initialization of the array.

For higher dimensional arrays, we can omit the outermost dimension because the compiler can determine the dimension automatically. For example, the following array declaration

```
int array[][3][2]= { {{1,1}, {0,0}, {1,1}},
                     {{0,0}, {1,2}, {0,1}} };
```

creates a [2][3][2] dimensioned array.

However, the statement

```
int wrong_array[][] = { 1,2,3,4 };
```

is incorrect, as the compiler is unable to determine the size of the array.

Operations on Multidimensional Arrays

Program 9.11 gives a sample program on determining the sum of rows and columns of two-dimensional arrays. When accessing two-dimensional arrays, we use an index variable **row** to refer to the row number and another index variable **column** to refer to the column number. A nested **for** loop is used to access the individual elements of the array. To process the sum of rows, we use the index variable **row** as the outer **for** loop. To process the sum of columns, we use the index variable **column** as the outer **for** loop.

Program 9.11 Processing two-dimensional arrays.

```
#include <stdio.h>
main(void)
{
    int array[3][3]= {
                        {5, 10, 15},
                        {10, 20, 30},
                        {20, 40, 60}
                      };
    int row, column, sum;
```

```
for (row = 0; row < 3; row++)          /* sum of rows */
{
    sum = 0;
    for (column = 0; column < 3; column++)
        sum += array[row][column];
    printf("The sum of elements in row %d is %d\n", row+1,
        sum);
}
for (column = 0; column < 3; column++)
    /* sum of columns */
    {
        sum = 0;
        for (row = 0; row < 3; row++)
            sum += array[row][column];
        printf("The sum of elements in column %d is %d\n",
            column+1, sum);
    }
return 0;
}
```

Program output

```
The sum of elements in row 1 is 30
The sum of elements in row 2 is 60
The sum of elements in row 3 is 120
The sum of elements in column 1 is 35
The sum of elements in column 2 is 70
The sum of elements in column 3 is 105
```

Program 9.12 gives another example on processing matrix multiplication using two-dimensional arrays. The program multiplies matrix **A[3][3]** by matrix **B[3][3]**. The result is stored in matrix **C[3][3]**. A set of nested **for** loop is used to generate row and column indexes when computing individual element of the resulting matrix.

Program 9.12 Matrix multiplication using two-dimensional arrays.

```
#include <stdio.h>
main(void)
{
    int A[3][3] = { {1, 2, 3},
                    {2, 3, -1},
                    {3, -1, 2}};
    int B[3][3] = { {1, 2, 3},
```

```

                                {5, 7, 9},
                                {9, 11, 13}};
int C[3][3];
int l, m, n;

for (l = 0; l < 3; l++)
    for (m = 0; m < 3; m++) {
        C[l][m] = 0;
        for (n = 0; n < 3; n++)
            C[l][m] = C[l][m] + A[l][n]*B[n][m];
    }
printf("The product of the arrays is: \n");
for (l = 0; l < 3; l++) {
    for (m = 0; m < 3; m++)
        printf("%5d", C[l][m]);
    printf("\n");
}
return 0;
}

```

Program output

```

The product of the arrays is:
    38    49    60
     8    14    20
    16    21    26

```

9.9 Multidimensional Arrays and Pointers

Multidimensional arrays are also stored sequentially in the memory. For example, consider the following two-dimensional array:

```

int ar[4][2];    /* ar is an array of 4 elements; each */
                 /* element is an array of 2 integers */

```

where **ar** is also the address of the first element of the array. Figure 9.11 illustrates the two-dimensional array. In this case, the first element is an array of 2 integers. **ar** is the address of a two-integer sized object. We have

```

ar == &ar[0]           *ar == ar[0]
ar + 1 == &ar[1]       *(ar + 1) == ar[1]
ar + 2 == &ar[2]       *(ar + 2) == ar[2]
ar + 3 == &ar[3]       *(ar + 3) == ar[3]

```

`ar[0]` is an array of 2 integers, so `ar[0]` is the address of an integer sized object. Therefore, we have

```
ar[0] == &ar[0][0]          *ar[0] == ar[0][0]
```

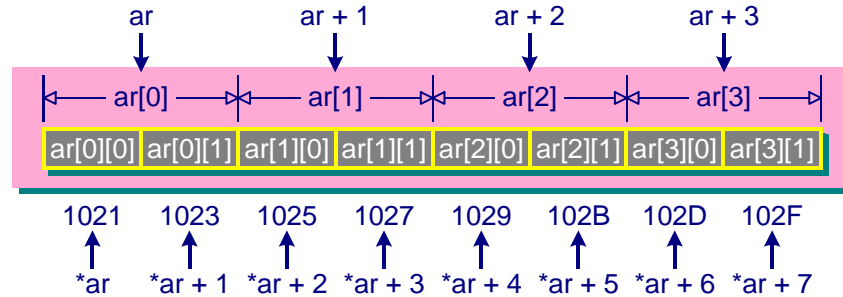


Figure 9.11 Two-dimensional array and pointers.

```
ar[1] == &ar[1][0]          *ar[1] == ar[1][0]
ar[2] == &ar[2][0]          *ar[2] == ar[2][0]
ar[3] == &ar[3][0]          *ar[3] == ar[3][0]
```

Note that adding 1 to a pointer or address yields a value larger by the size of the referred-to object. For example, although `ar` has the same address value as `ar[0]`, `ar+1` (i.e. 1025) is different from `ar[0]+1` (i.e. 1023). This is due to the fact that `ar` is a two-integer sized object, while `ar[0]` is an integer sized object. Adding 1 to `ar` increases by 4 bytes. `ar[0]` refers to `*ar`, which is the address of an integer, adding 1 to it increases by 2 bytes.

We can use the `*` operator to dereference a pointer or an address to yield the value represented by the referred-to object:

```
*(ar[0]) == ar[0][0]
and  *ar == ar[0]
=>  **ar == *(ar[0]) == ar[0][0]
```

This is called *double indirection*.

In general, we can represent individual elements of a two-dimensional array as

```
ar[m][n] == *( *(ar + m) + n )
```

where `m` is the index associated with `ar`, and `n` is the index associated with the subarray `ar[m]`. When `n` is added to `*(ar+m)`, i.e. `*(ar+m)+n`, it becomes the address of the element of `ar[m][n]`. Applying the `*` operator to that address gives the content at that address location.

9.10 Multidimensional Arrays as Function Arguments

The individual element of a two-dimensional array can be passed as an argument to a function. This can be done by specifying the array name with the corresponding row number and column number.

If an entire multidimensional array is to be passed as an argument to a function, this can be done in a similar manner to a one-dimensional array. The definition of a function with a two-dimensional array argument is given as

```
void function(int array[2][4])  
or void function(int array[][4])
```

The first dimension (i.e. the row information) of the array can be excluded because C compiler can determine the first dimension automatically. However, the number of columns must be specified. This is illustrated by the following example. The assignment statement

```
array[1][3] = 100;
```

requests the compiler to compute the address of **array[1][3]** and then places a value of 100 to that address. In order to compute the address, the dimension information must be given to the compiler. Let us redefine **array** as:

```
int array[D1][D2];
```

The address of **array[1][3]** is computed as:

```
baseAddress + row * D2 + column  
=> baseAddress + 1 * 4 + 3  
=> baseAddress + 7
```

where the baseAddress is the address pointing to the beginning of **array**. Notice **D1** is not needed in computing the address. Therefore, we can omit the value of the first dimension of an array in defining a function, which takes arrays as its formal arguments. Figure 9.12 illustrates the concept.

Assume that **function()** has been defined with the parameter **table** which is a two-dimensional array:

```
int table[2][4];
```

The statements

```
int ar2[2][4];  
....
```



```
function(ar2);
```

in the `main()` function call `function()`. Similar to the one-dimensional array,

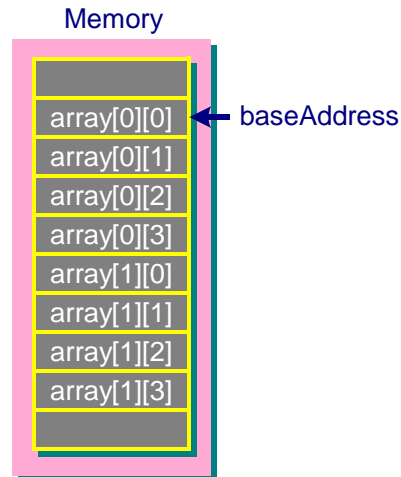


Figure 9.12 Individual element address in a two-dimensional array.

the name of the array `ar2` is specified as the argument without any subscripts in the function call.

The function prototype of the function becomes

```
void function(int table[2][4]);  
or void function(int table[][4]);
```

The above discussion and definition of two-dimensional arrays can be generalized to the arrays of higher dimensions.

Program 9.13 gives a sample program on determining the total sum of all the rows and the total sum of all the columns of a two-dimensional array. Two functions `sum_rows()` and `sum_columns()` are written to compute the total sums. Both functions take an array as its argument:

```
int sum_rows(int ar[][3])  
int sum_columns(int ar[][3])
```

When calling the functions, the name of the array is passed to the calling functions:

```
total_row = sum_rows(array);  
total_column = sum_columns(array);
```

The total values are computed in the two functions and placed in the two variables `total_row` and `total_column` respectively.

Program 9.13 Passing two-dimensional arrays as arguments.

```
#include <stdio.h>
int sum_rows(int ar[][3]);
int sum_columns(int ar[][3]);
main(void)
{
    int array[3][3]= {
        {5, 10, 15},
        {10, 20, 30},
        {20, 40, 60}
    };

    int total_row, total_column;
    total_row = sum_rows(array);
    total_column = sum_columns(array);
    printf("The sum of all elements in rows is %d\n",
        total_row);
    printf("The sum of all elements in columns is %d\n",
        total_column);
    return 0;
}
int sum_rows(int ar[][3])
{
    int row, column;
    int sum=0;
    for (row = 0; row < 3; row++)
    {
        for (column = 0; column < 3; column++)
            sum += ar[row][column];
    }
    return sum;
}
int sum_columns(int ar[][3])
{
    int row, column;
    int sum=0;
    for (column = 0; column < 3; column++)
    {
        for (row = 0; row < 3; row++)
            sum += ar[row][column];
    }
    return sum;
}
```

```
}

```

Program output

```
The sum of all elements in rows is 210
The sum of all elements in columns is 210
```

Program 9.14 gives another example on processing matrix multiplication using two-dimensional arrays as function arguments. A function `matrix_multiply()` has been written to perform multiplication operation on two two-dimensional matrices, **A1** and **B1**. The function uses input arrays as its arguments. The resulting matrix **C1** is passed back through call by reference. We can use the function call

```
matrix_multiply(A1, B1, C1);
```

to call the `matrix_multiply()` function, where **A1** and **B1** are input matrices and **C1** is the resulting matrix.

Program 9.14 Matrix multiplication using two-dimensional arrays as arguments.

```
#include <stdio.h>
void matrix_multiply(int A[][3], int B[][3], int C[][3]);
main(void)
{
    int A1[3][3] = { {1, 2, 3},
                     {2, 3, -1},
                     {3, -1, 2}};
    int B1[3][3] = { {1, 2, 3},
                     {5, 7, 9},
                     {9, 11, 13}};

    int C1[3][3];
    int l, m;

    matrix_multiply(A1, B1, C1);
    printf("The product of the arrays is: \n");
    for (l = 0; l < 3; l++) {
        for (m = 0; m < 3; m++)
            printf("%5d", C1[l][m]);
        printf("\n");
    }
    return 0;
}
```

```
void matrix_multiply(int A[][3], int B[][3], int C[][3])
{
    int l, m, n;
    for (l = 0; l < 3; l++)
        for (m = 0; m < 3; m++) {
            C[l][m] = 0;
            for (n = 0; n < 3; n++)
                C[l][m] = C[l][m] + A[l][n] * B[n][m];
        }
}
```

Program output

The product of the arrays is:

38	49	60
8	14	20
16	21	26

Applying a One-Dimensional Function to Two-Dimensional Arrays

A function that is written for processing one-dimensional arrays can be used to deal with two-dimensional arrays. This is illustrated in Program 9.15.

In Program 9.15, **array** is an array of 2x4 integers. Three functions **display1()**, **display2()** and **display3()** have been written to access the elements of the array with the specified **size** and prints out the contents. In **display1()**, it accepts a pointer variable and accesses the elements of the array using the pointer variable. In **display2()**, it accepts the array pointer and uses array index to access the elements of the array. In **display3()**, it also accepts an array pointer and accesses the elements of the two-dimensional array using the row and column indexes for a two-dimensional array.

Program 9.15 Processing two-dimensional arrays as one-dimensional arrays.

```
#include <stdio.h>
void display1(int *ptr, int size);
void display2(int ar[], int size);
void display3(int ar[][4], int size);

main(void)
{
    int array[2][4] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int i;
```

```
    for (i=0; i<2; i++) {
        display1(array[i], 4);
        display2(array[i], 4);
    }
    display3(array, 2);
    display1(array, 8);
    display2(array, 8);
    return 0;
}

void display1(int *ptr, int size)
{
    int j;
    printf("Display1 result: ");
    for (j=0; j<size; j++)
        printf("%d ", *ptr++);
    putchar('\n');
}

void display2(int ar[], int size)
{
    int k;
    printf("Display2 result: ");
    for (k=0; k<size; k++)
        printf("%d ", ar[k]*5);
    putchar('\n');
}

void display3(int ar[][4], int size)
{
    int i,j;
    printf("Display3 result: ");
    for (i=0; i<size; i++)
        for (j=0; j<4; j++)
            printf("%d ", ar[i][j]*10);
    putchar('\n');
}
```

Program output

```
Display1 result: 0 1 2 3
Display2 result: 0 5 10 15
Display1 result: 4 5 6 7
Display2 result: 20 25 30 35
Display3 result: 0 10 20 30 40 50 60 70
Display1 result: 0 1 2 3 4 5 6 7
Display2 result: 0 5 10 15 20 25 30 35
```

In the first **for** loop of the **main()** function, when **i=0**, we pass **array[0]** to **display1()** and **display2()**. **array[0]** corresponds to the address of **array[0][0]** (i.e. **&array[0][0]**). The two functions then access the array starting from the location **array[0][0]**, and prints the 4 elements out to the display as specified in the function. When **i=1**, **array[1]** is passed to **display1()** and **display2()**. Now, **array[1]** corresponds to the address of **array[1][0]** (i.e. **&array[1][0]**). The two functions then access the 4 elements starting from **array[1][0]**, and print the contents of the 4 elements. This is illustrated in Figure 9.13.

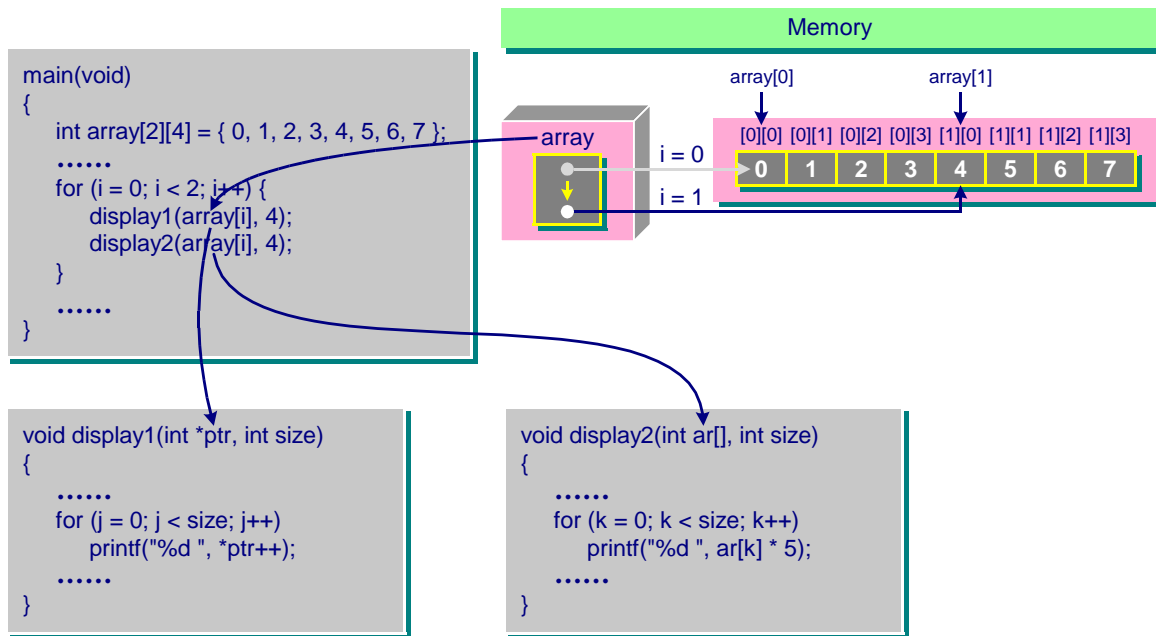


Figure 9.13 Processing two-dimensional arrays as one-dimensional arrays.

When the following function

```
display3(array, 2);
```

is called, the value in the **size** argument (i.e. 2) is used to indicate the number of rows in the array. Same as accessing other typical two-dimensional arrays, the row and column indexes are used in this function to process individual elements of the two-dimensional array, and print out the contents to the display.

We can also view **array** as an array of 8 integers. When we pass **array** as an argument to the functions **display1()** and **display2()** in

```
display1(array, 8);  
display2(array, 8);
```

the pointers `ptr` and `ar` in `display1()` and `display2()` are initialized to the address of `array[0][0]`. As a result, `*ptr` and `ar[0]` correspond to `array[0][0]`, while `*(ptr+1)` and `ar[1]` correspond to `array[0][1]`, and so on. Similarly, `*(ptr+4)` and `ar[4]` correspond to `ar[4][0]`. Therefore, all the elements of the two-dimensional array can be accessed and printed to the display.

The sizeof Operator and Arrays

sizeof is an operator which gives the size (in bytes) of its operand. Its syntax is

```
sizeof(operand)  
or  sizeof operand
```

The **operand** can either be a type enclosed in parenthesis or an expression. We can also use it with arrays. This is illustrated in Program 9.16.

In Program 9.16, the **sizeof** operator returns the number of bytes of the array. The second **sizeof** operator returns the number of bytes of each element in the array. Therefore, the number of elements can be calculated by dividing the size of the array by the size of each element in the array.

Program 9.16 Calculating the number of array elements using the sizeof operator.

```
#include <stdio.h>  
main(void)  
{  
    int array[2][4];  
  
    printf("Array size is %d\n", sizeof(array) / sizeof(  
        array[0][0]));  
    return 0;  
}
```

Program output

```
Array size is 8
```

In Program 9.17, when we apply **sizeof** to the array **item**, we have the array size. The number 20 is obtained by multiplying the number of elements in **item** (i.e. 5) by the number of bytes per element (i.e. 4 bytes). But when we apply **sizeof** to the pointer variable **a**, we only obtain the size of the pointer (i.e. 4 bytes).

Program 9.17 Using sizeof to calculate the size of a pointer and an array element.

```
#include <stdio.h>
#define SIZE 5
int sum(int [], int);
int item[SIZE]={1,2,3,4,5};
main(void)
{
    int total;
    total = sum(item, SIZE);
    printf("Size of item = %d\n", sizeof item);
    return 0;
}
int sum(int a[], int n)
{
    int i, s=0;
    printf("Size of a = %d\n", sizeof a);
    for (i=0; i<n ; i++)
        s+=a[i];
    return s;
}
```

Program output

```
Size of a = 4
Size of item = 20
```

9.11 Exercises

1. A company employs 20 workers. Each worker works five days a week from Monday to Friday. A two-dimensional array of integers is declared globally as

```
int production[20][5];
```


This array is used to store the production output for each worker. For example, `production[2][1]` indicates the production output for worker with identity 2 on Tuesday.

- (a) Write a function to read the production output of all workers from `stdin`. The input contains only the production output of all the workers for all work days. The input is then stored in the array `production`. The function prototype is given as: `void ReadInput(void);`
- (b) Write a function to return the weekly average production output of all the workers. The function prototype is given as: `float ComputeAverage(void);`
- (c) Write a function to identify the most productive worker for a week. The function should print a line on the screen: "Worker x is the most productive worker", where x indicates the worker identity. The function prototype is given as: `void FindtheBest(void);`

2. Consider the following unknown function.

```
void unknown(int ar1[ ], int b[ ], int c[ ], int m, int
              n)
{
    int i = 0, j = 0, k = 0;

    while ( i < m && j < n )
        if ( ar1[i] < ar2[j] )
            ar3[k++] = ar1[i++];
        else
            ar3[k++] = ar2[j++];
    while ( i < m )
        ar3[k++] = ar1[i++];
    while ( j < n )
        ar3[k++] = ar2[j++];
}
```

where `ar1[]` and `ar2[]` are arrays of sizes `m` and `n` respectively. Assume that an array `ar3[]` of appropriate size has been declared. Determine the purpose of the function. Illustrate how the function works by applying it to the following input data: `ar1[m] = { 0, 1, 3, 5, 7, 9, 11}` and `ar2[n] = { -3, -1, 1, 2, 7, 8, 9}`.

3. Write a recursive C function `reverseAr()` which can print the array of integers in reverse order. For example, if `ar[5] = {1, 2, 3, 4, 5}`, then the output 5, 4, 3, 2, 1 will be printed on the display after applying the function `reverseAr(ar, 5)`. The function prototype is given as follows:

```
void reverseAr(int ar[ ], int size);
```

4. Write both the iterative and recursive versions of a function `find()` that returns the subscript of the first appearance of a number in the array. For example, if the array is `{3,6,9,4,7,8}`, then `find(6, array, 3)` will return 0 where 6 is the size of the array and 3 is the number to find, and `find(6, array, 9)` will return 2. If the required number is not in the array, the function will return -1. The function prototype is given as follows:

```
int find(int size, int array[], int number);
```



Character Strings and String Functions

In addition to handling numerical data, programs are also required to deal with alphabetic data. Strings are arrays of characters. C libraries provide a number of functions for performing operations on strings. In this chapter, string constants and string variables are first introduced. The different commonly used string functions from C libraries are discussed.

This chapter covers the following topics:

- 10.1 String Constants
- 10.2 String Variables
- 10.3 String Input
- 10.4 String Output
- 10.5 String Functions
- 10.6 The ctype.h Character Functions
- 10.7 String to Number Conversions
- 10.8 Formatted String Input and Output
- 10.9 Arrays of Character Strings
- 10.10 Command Line Arguments

10.1 String Constants

A string is an array of characters terminated by a null character, `'\0'`. A string constant is a series of characters in double quotes:

"C Programming"

When the compiler encounters a string constant, it allocates space for each individual character of the string and adds a terminating null character at the end of the string. A pointer pointing to its first character is returned. Typically, we can assign the pointer of the string constant to a pointer variable:

```
char *str = "C Programming";
```

Figure 10.1 shows the string constant representation with a pointer variable.



Figure 10.1 A string.

A string constant is essentially a pointer to the first character of an array of characters. We can use **#define** to define string constants:

```
#define NTU "Nanyang Technological University"
```

String constants can also be used as function arguments for **printf()** and **puts()** functions:

```
printf("Welcome to Introduction to C Programming.\n");
puts("Welcome to Introduction to C Programming.");
```

It is important to notice the difference between the character constant **'x'** and a string constant **"x"**. The character constant **'x'** consists of a single character of data type **char**, while the character string constant **"x"** consists of two characters, i.e. the character **'x'** and the null character **'\0'**, and is an array of type **char**.

10.2 String Variables

A string is an array of characters. We can define strings using array notation or pointer notation.

String and Array

To declare strings using array notation, we can use the following declarations:

```
char str[] = "some text";  
char str[10] = "yes";
```

However, the following declaration

```
char str[4] = "four";
```

is incorrect. The array `str` only allocates 4 elements to hold its data, which is not enough as it needs an additional element to hold the null character `'\0'` that is automatically added at the end of the sequence of characters. We can also create strings as follows:

```
char str[] = {'a','b','c','\0'};
```

An array of four elements of type `char` is created. However, it is tedious to initialize array by listing all the characters in braces. We can simply enclose the characters using double quotes as if they were a string constant. This is equivalent to

```
char str[] = "abc";
```

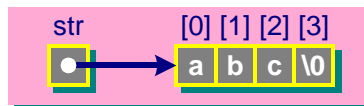


Figure 10.2 String and array.

Figure 10.2 shows the data stored in the string `str`. As a string constant returns a pointer to its first character, the array name `str` contains the address of the first element of the array as other kinds of arrays:

```
str == &str[0];  
*str == 'a';  
*(str+1) == str[1] == 'b';
```

However, the following statements

```
str++;  
str--;
```

are invalid. As `str` is a pointer constant, we cannot change its value.

String and Pointer

We can assign a string constant to a pointer that points to type **char**:

```
char *ptr = "This is a string";
```

When a string constant is assigned to a pointer, the C compiler will allocate memory space to hold the string constant, store the starting address of the string (i.e. the address of the character 'T') in the pointer, and terminate the string with a null character. However, **ptr** is a pointer variable that can be changed. For example, the statement

```
ptr++;
```

changes the **ptr** variable to point to the next character (i.e. 'h') in the string "This is a string". This is illustrated in Figure 10.3.

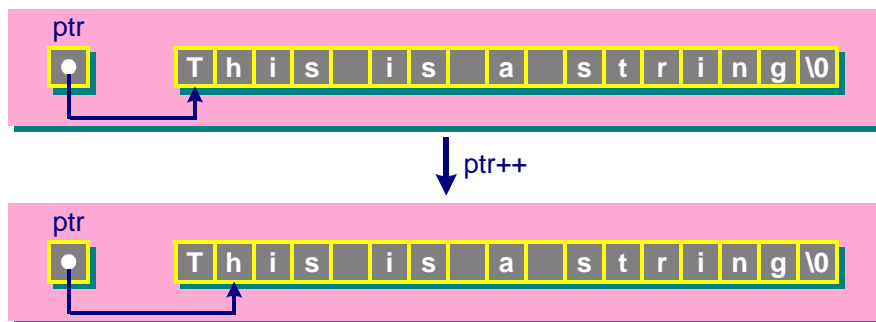


Figure 10.3 String and pointer.

It is important to distinguish the declaration of string variables that use array or pointer. For example, the following strings are defined as:

```
char str1[] = "How are you?"; /* using array */  
char *str2 = "How are you?"; /* using pointer */
```

The **str1** declaration creates an array of type **char**. The array has been allocated with memory to hold 13 elements including the null character. The C compiler also creates a pointer constant that is initialized to point to the first element of the array, **str1[0]**. In the **str2** declaration, the C compiler creates a pointer variable **str2** that points to the first character of the string. It contains the memory address of the first character of the string 'H'. Both **str1** and **str2** are pointers. However, the difference between the two declarations is that **str1** is a pointer (or address)

constant, while **str2** is a pointer variable. Pointer constant means that the value cannot be changed, while pointer variable allows its value to be changed. Therefore, the following statements

```
++str2;
str2 = str1;
```

are valid, while the following statements

```
++str1;
str1 = str2;
```

are invalid.

Program 10.1 gives an example on using array and pointer to declare strings. Figure 10.4 shows the memory locations that the pointers are pointing at.

Program 10.1 Using arrays and pointers for strings.

```
main(void)
{
    char array[14];
    char *ptr1 = "14 characters", *ptr2;

    ptr1[9] = 'B';           /* valid */
    ptr1 = "8 chars";        /* valid */
    ptr1[9] = 'B';           /* invalid */
    ptr2[9] = 'B';           /* invalid */
    *ptr2 = "invalid";       /* invalid */
    array = "invalid";       /* invalid */
    array[9] = 'B';          /* valid */
    ptr1 = array;            /* valid */
    ptr1 = "A new string";   /* valid */
    return 0;
}
```

The declaration

```
char array[14];
```

declares an array of type **char** called **array[]** that contains 14 elements. The declaration

```
char *ptr1 = "14 characters", *ptr2;
```

creates two pointer variables **ptr1** and **ptr2**. The pointer variable **ptr1** is

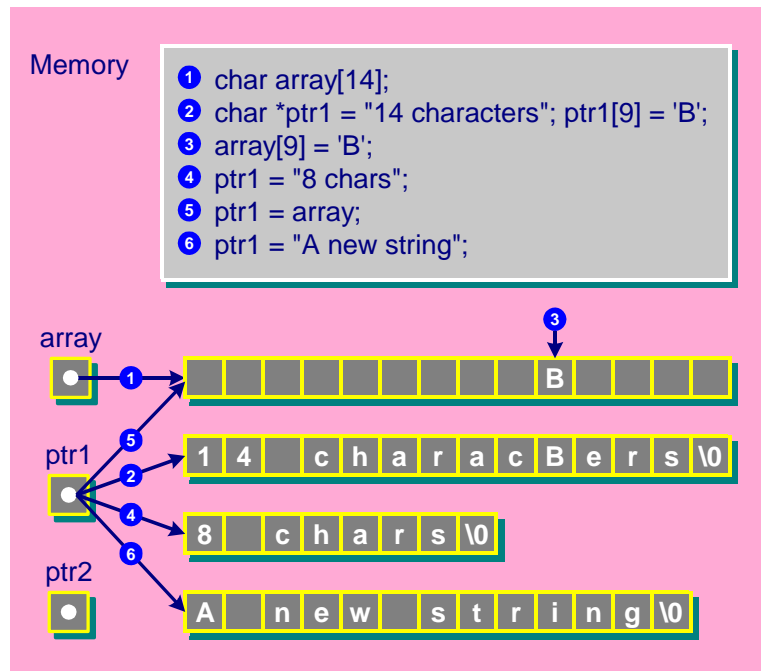


Figure 10.4 Arrays and pointers of strings.

initialized and pointed to the string **"14 characters"**. The statement

```
ptr1[9] = 'B';           /* valid */
```

is valid and the string is updated accordingly. The statement

```
ptr1 = "8 chars";       /* valid */
```

will update **ptr1** to point to a new string **"8 chars"**. However, the statement

```
ptr1[9] = 'B';          /* invalid */
```

is invalid because **ptr1** only points to an array of 8 elements. There is no storage available in the array to store the character **'B'**. The statement

```
ptr2[9] = 'B';          /* invalid */
```

is invalid because **ptr2** is not initialized yet, and no memory is allocated to store the character **'B'**. The statement

```
*ptr2 = "invalid";      /* invalid */
```


is invalid because type mismatch occurs, as the left-hand side is of data type **char**, while the right-hand side is an array of type **char**. The statement

```
array = "invalid";          /* invalid */
```

is invalid because the left-hand side is an array address, while the right-hand side is a string pointer. We cannot change the base address of an array. The statement

```
ptr1 = array;               /* valid */
```

changes the pointer variable **ptr1** to point to the same address contained in **array**. The statement

```
ptr1 = "A new string";     /* valid */
```

is valid which changes the pointer variable **ptr1** to point to the new string "A new string".

Program 10.2 illustrates the difference between array and pointer notations for processing strings. The function **length1()** uses the array notation and the function **length2()** uses the pointer notation. Both versions measure the length of strings.

Program 10.2 Determining the length of a string.

```
#include <stdio.h>
main(void)
{
    char word[] = "abc";
    char *wordptr = "programming";
    int length1(char []);
    int length2(char *);

    printf("The length is (using array): %d, %d\n",
           length1(word), length1(wordptr));
    printf("The length is (using pointer): %d, %d\n",
           length2(word), length2(wordptr));

    return 0;
}

/* First version that uses array notation */
int length1(char string[])
{
    int count = 0;
    while (string[count] != '\0')
```

```

    count++;
    return(count);
}

/* Another version that uses pointer notation */
int length2(char* string)
{
    int count = 0;

    while (*(string + count))
        count++;
    return(count);
}

```

Program output

The length is (using array): 3, 11
 The length is (using pointer): 3, 11

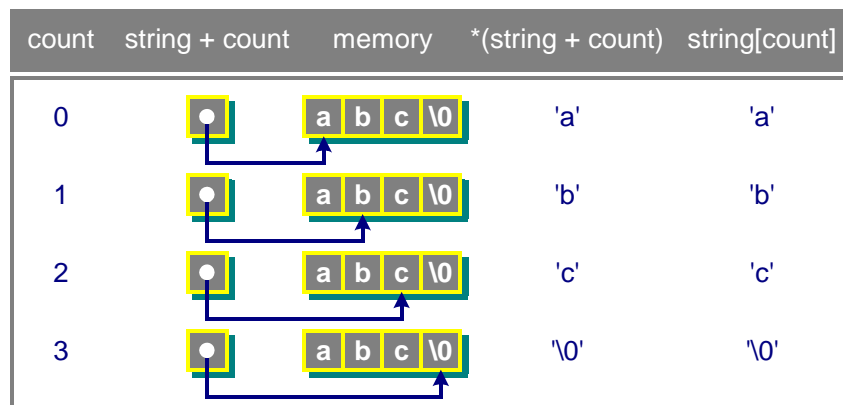


Figure 10.5 Using a pointer to access a string.

The declaration

```
char word[] = "abc";
```

creates an array of type **char** called **word[]**. This array contains four elements including the null character. The declaration

```
char *wordptr = "programming";
```

creates a pointer variable called **wordptr** that points to the first character of the string "programming".

In the first version, the function **length1()** uses the statement

```
while (string[count] != NULL)
```

to check for the null character ('\0') in the **while** loop while measuring the length of the string. In the second version, the function **length2()** uses the statement

```
while (*(string + count))
```

to check for terminating null character. The expression ***(string + count)** first gets the character stored at the address location contained in **(string + count)**, and then tests whether it is a null character. For any characters other than the null character, the condition will be true, and the statements in the body of the **while** loop will be executed to measure the length of the string. This is illustrated in Figure 10.5.

10.3 String Input

Two steps are involved in order to read a string into a program:

1. enough storage space should first be allocated to store the string;
2. an input function is then used to fetch the string.

Creating Memory Space

To create space for a string, we may include an explicit array size as shown in the following declaration:

```
char str[81];
```

This declaration statement creates a character array **str** of 81 elements. Once the storage space has been acquired, we can read in the string through the C library functions. Two commonly used C library input functions are **gets()** and **scanf()**.

The gets() Function

The **gets()** function gets a string from the standard input device. It reads characters until it reaches a newline character (**\n**). A newline character is generated when the **<Enter>** key is pressed. The **gets()** function reads all the characters up to and including the newline character, replaces the newline

character with a null character and passes them to the calling function as a string. The function prototype of `gets()` is

```
char *gets(char *ptr);
```

The `gets()` function returns a null pointer if it fails, otherwise the same pointer `ptr` is returned. It is important to make sure that sufficient storage is allocated to hold the input string. The null pointer contains a null address that is the symbolic constant `NULL` defined in the header file `stdio.h`. This is different from the null character (`'\0'`) in that the null character is a data object of type `char` with ASCII value 0, whereas the null pointer contains an address. In addition, the size of the null character is 1 byte, while the null pointer is 4 bytes long.

Program 10.3 reads in a string and prints the string to the screen. It is important to ensure that the array size of `name` is big enough to hold the input string. Otherwise, the extra characters can overwrite the adjacent memory variables.

Program 10.3 Reading strings using `gets()`.

```
#include <stdio.h>
main(void)
{
    char name[81]; /* allocate storage space */

    printf("Hi, what is your name?\n");
    gets(name);
    printf("Hello, %s.\n", name);
    return 0;
}
```

Program input and output

```
Hi, what is your name?
Hui Siu Cheung
Hello, Hui Siu Cheung.
```

The `scanf()` Function

The `scanf()` function with the `%s` format specification can be used to read in a string. This function will return `EOF` if it fails, otherwise the number of items read by `scanf()` will be returned. An example of the `scanf()` function to read in a string is given as follows:

```
scanf("%s", str);
```

Unlike other data input, there is no need to have the address-of (&) operator to be placed before the name of the array. This is due to the fact that the array name is the address of the first element of the array.

It is important to ensure that sufficient storage is allocated to hold the string. If more characters are read in than the storage space allocated to hold the string, the additional characters will be overwritten to the adjacent memory locations. For example, consider the following declaration statements:

```
char *str = "string space";
scanf("%s", str);
```

A string pointer constant is created, and 13 bytes of storage are created to hold the string constant. If the string read in by the `scanf()` function is longer than 13 bytes, it will be overwritten to the adjacent memory locations.

There are two ways to read input string in `scanf()`. We may use either a `%s` or `%ns` conversion specifier, where `n` is the field width specification. In both cases, reading starts at the first nonwhitespace character. When `%ns` is used, `scanf()` reads up to `n` input characters or when the first whitespace character (i.e. blank, tab or newline) is encountered. For example, `%6s` will stop after the first six characters are read. When `%s` is specified, the `scanf()` function reads the string up to the next whitespace character.

Program 10.4 illustrates the `scanf()` function to read names from the input. Figure 10.6 shows the processing of the `scanf()` function. For example, when the input "Siu Cheung Hui" is entered. Then, `name1 = "Siu"` as the first whitespace is encountered after the character 'u' before the specified number 5. `Name2 = "Cheun"` as the specified number of characters is met after reading the character 'n'. `Name3 = "g"` as a whitespace is encountered after the character 'g' and the reading of the input buffer stops.

Program 10.4 Reading strings using `scanf()`.

```
#include <stdio.h>
main(void)
{
    static char name1[20], name2[20], name3[20];
    int count;

    printf("Please enter your names. \n");
    count = scanf("%5s %5s %6s", name1, name2, name3);
    printf("I read %d names %s %s %s.\n", count, name1,
        name2, name3);
    return 0;
}
```

Program input and output

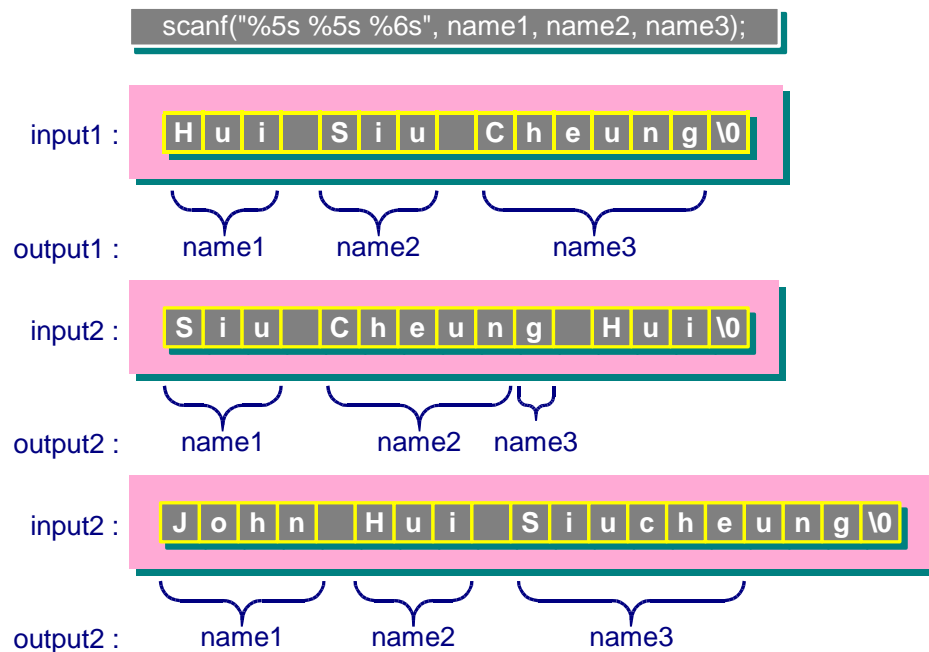
```

Please enter your names.
Hui Siu Cheung
I read 3 names Hui Siu Cheung.

Please enter your names.
Siu Cheung Hui
I read 3 names Siu Cheun g.

Please enter your names.
John Hui SiuCheung
I read 3 names John Hui SiuChe.

```

**Figure 10.6** Processing of input by `scanf()`.

The `gets()` function differs from the `scanf()` function in that `gets()` reads an entire line up to the first newline character, and it also stores any characters, including whitespace, up to the first newline character. The `scanf()` function is less convenient to use than the `gets()` function for reading string input. The `gets()` function is also faster than the `scanf()` function.

10.4 String Output

The two commonly used standard library functions for printing strings are `puts()` and `printf()`.

The `puts()` Function

The function prototype for the `puts()` function is

```
int puts(const char *ptr);
```

It prints a string on the standard output device. A newline character is automatically added to the end of the string. Thus, the newline character is printed after the string. `EOF` is returned if `puts()` fails, otherwise the number of characters written will be returned.

Program 10.5 reads in a string using the `gets()` function and prints the string on the screen using the `puts()` and `printf()` functions.

The `printf()` Function

The function prototype for the `printf()` function is

```
printf(control-string, argument-list);
```

It prints formatted output on the standard output device. It returns a negative value if `printf()` fails, otherwise the number of characters printed will be returned. It differs from the `puts()` function in that no newline is added at the end of the string. The `printf()` function is less convenient to use than the `puts()` function. However, the `printf()` function provides the flexibility to the user to control the format of the data to be printed.

Program 10.5 Printing strings using `puts()` and `printf()`.

```
#include <stdio.h>
main(void)
{
    char str[81];

    printf("What is your name: ");
    if (gets(str) == NULL) {
        printf("Error\n");
    }
    puts(str);
    printf("How are you? %s\n", str);
}
```

```
    return 0;
}
```

Program input and output

```
What is your name: Siu Cheung Hui
Siu Cheung Hui
How are you? Siu Cheung Hui
```

10.5 String Functions

The standard C library provides many functions that perform string operations. To use any of these functions, we must include the header file *string.h* in the program:

```
#include <string.h>
```

String functions are very useful for writing programs that involve the manipulation of strings. Some examples of string functions include finding the length of strings, combining two strings, comparing two strings, and searching a string for a specific character. Programmers are encouraged to use these functions instead of developing their own functions. Table 10.1 lists some of the more commonly used string functions provided in *string.h*. In this section, we describe some of the most useful C string handling functions. These include **strlen()**, **strcmp()**, **strcpy()**, **strcat()**, **strchr()** and **strrchr()**.

Table 10.1 C string functions.

Function	Meaning	Description
strcat()	string concatenation	appends one string to another
strncat()	string concatenation of n characters	appends a portion of a string to another string
strchr()	string has character	finds the first occurrence of a specified character in a string
strrchr()	string has character (search from end)	finds the last occurrence of a specified character in a string
strstr()	string has substring	finds the position of the first character of one string in another
strcmp()	string comparison of n characters	compares two strings

strncmp()	string comparison of n characters	compares two strings up to a specified number of characters
strcpy()	string copy	copies a string to an array
strncpy()	string copy of n characters	copies a portion of a string to an array
strpbrk()	string pointer break	finds the first occurrence of any specified characters in a substring within a string
strlen()	string length	computes the length of a string

The strlen() Function

The **strlen()** function finds the length of a string. The function prototype of **strlen()** is

```
unsigned strlen(const char *str);
```

strlen() computes the length of the string pointed to by **str**. It returns the number of characters that precede the terminating null character. The null character is excluded in the calculation. Program 10.6 illustrates the use of the **strlen()** function. The program creates a character array called **line[]** to store the string. The character array **line[]** is initialized to the character string constant **"This is a string"**. The length of this string is then calculated using the **strlen()** function, and printed on the display.

The **sizeof** operator can also be used to determine the number of characters in a string. However, this function includes the terminating null character in its calculation.

Program 10.6 Finding the length of a character string.

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

main(void)
{
    char line[81] = "This is a string";
    printf("The length of the string is %d.\n",
        strlen(line));
    return 0;
}
```

Program output

The length of the string is 16.

The strcat() and strncat() Functions

The **strcat()** function concatenates two strings to form a new string. The function prototype of **strcat()** is

```
char *strcat(char *str1, const char *str2);
```

strcat() appends a copy of the string pointed to by **str2** to the end of the string pointed to by **str1**. The initial character of **str2** overwrites the null character at the end of **str1**. **strcat()** returns the address value of **str1**. **str2** is unchanged. This is illustrated in Figure 10.7. For example, two strings are declared and initialized as follows:

```
char str1[16] = "Problem ";  
char str2[8] = "Solving";
```

After the function

```
strcat(str1, str2);
```

is executed, the **str2** is unchanged and **str1** has been changed as shown in Figure 10.7.

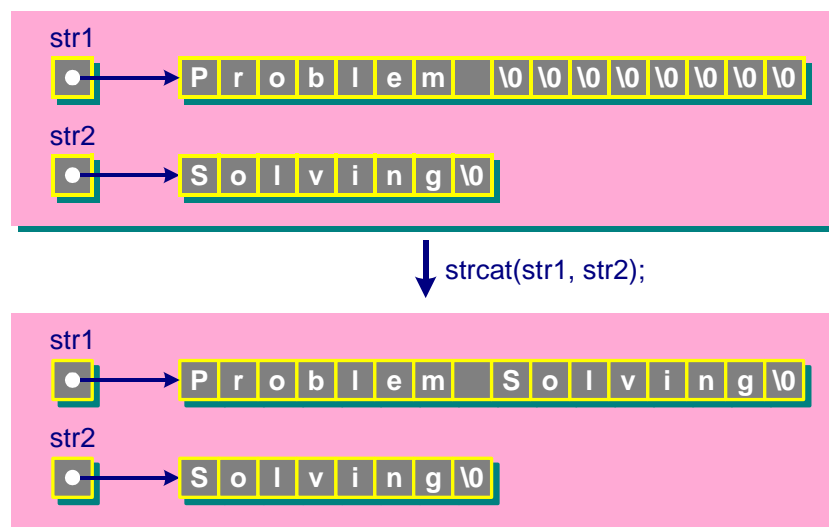


Figure 10.7 String concatenation.

Program 10.7 illustrates the **strcat()** function. It is important to note that the storage allocated to **str1** should be big enough to hold the concatenated string as **strcat()** will not check the storage requirement before performing the concatenation operation.

Program 10.7 Concatenating two strings.

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

main(void)
{
    char str1[40] = "Problem ";
    char *str2 = "Solving";

    printf("The first string: %s\n", str1);
    printf("The second string: %s\n", str2);
    strcat(str1, str2);
    printf("The combined string: %s\n", str1);
    return 0;
}
```

Program output

```
The first string: Problem
The second string: Solving
The combined string: Problem Solving
```

The **strncat()** function works similarly to the **strcat()** function, except it needs a third argument **n**, an integer which specifies the number of characters to be added to form the new string. The concatenation process stops after **n** characters of **str2** are added to **str1**. A null character is then inserted to the end of the new string.

The strcmp() and strncmp() Functions

The **strcmp()** function compares the contents of two strings. The prototype of **strcmp()** is

```
int strcmp(const char *str1, const char *str2);
```

It compares the string pointed to by **str1** to the string pointed to by **str2**. It takes the two strings and performs a letter-by-letter, alphabetic order comparison. The

comparison is based on ASCII codes for the characters. It returns an integer greater than, equal to or less than zero, according to whether the string pointed to by **str1** is greater than, equal to or less than the string pointed to by **str2** respectively. Table 10.2 lists the return values from string comparison.

Table 10.2 Return values of string comparison.

Return value	Description
0	if the two strings are equal
> 0	if the first string follows the second string alphabetically, i.e. the first string is larger
< 0	if the first string comes first alphabetically, i.e. the first string is smaller

It is interesting to note that in ASCII codes, uppercase characters come before lowercase characters, and digits come before the letters. For example, 'a' has the ASCII code value of 97, while 'A' has 65. If the initial characters are the same, then the **strcmp()** function moves along the string until it finds the first pair of different characters, and returns the comparison result. For example, when comparing "abcde" with "abcd", the first four characters are the same in the two strings. But when it comes to the character 'e' in the first string, it will be comparing with the terminating null character with ASCII code value of 0 in the second string. The function then returns a positive value. This way of ordering strings is called *lexicographic order*.

Program 10.8 illustrates the **strcmp()** function. Generally, we are not interested in the actual values returned by **strcmp()**. The actual values to be returned depend on individual system implementation. When comparing "A" to "C", some systems return -1, others return -2, i.e. the difference in ASCII code values. However, in many applications, we are only interested to find out whether the two strings are equal or not.

Program 10.8 Comparing two strings.

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

main(void)
{
    char str1[81], str2[81];
    int result;

    printf("String Comparison:\n");
    str1[0] = 'A';
    while (str1[0]) {
```

```
    printf("Enter the first string: ");
    gets(str1);
    printf("Enter the second string: ");
    gets(str2);
    result = strcmp(str1, str2);
    printf("The result of the comparison is %d\n\n",
        result);
}
return 0;
}
```

Program input and output

```
String Comparison:
Enter the first string: A
Enter the second string: B
The result of the comparison is -1

Enter the first string: ABa
Enter the second string: ABA
The result of the comparison is 1

Enter the first string: A0
Enter the second string: A1
The result of the comparison is -1
```

The **strncmp()** function is similar to that of the **strcmp()** function except that it takes a third argument **n**, an integer, that specifies the number of characters to compare. A typical call has the format **strncmp(str1, str2, n)**. If the number **n** is greater than the length of the string, it works exactly the same as **strcmp()**. This function is very useful in comparing portions of strings. For example, when the function

```
strncmp("Apple", "Applepie", 5);
```

is executed, it returns 0 as the two strings are identical for the first 5 characters.

Program 10.9 shows another example which uses the **strcmp()** function. In this program, it reads a few lines from the standard input, reverses the characters, and writes each line to the standard output. The input terminates when the user enters "END".

Program 10.9 Using the strcmp() function.

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

```
#include <string.h>
void reverse(char *);

main(void)
{
    char line[132];
    gets(line);
    while (strcmp(line, "END")!=0) {
        reverse(line);
        printf("%s\n", line);
        gets(line);
    }
    return 0;
}

void reverse(char *s)
{
    char c, *end;
    end = s + strlen(s) - 1;
    while (s<end) {
        c = *s;
        *s++ = *end; /* i.e. *s = *end; s++; */
        *end-- = c;  /* i.e. *end = *c; end--; */
    }
}
```

Program input and output

```
how are you
uoy era woh
END
```

The strcpy() and strncpy() Functions

The **strcpy()** function copies one string to another. The function prototype of **strcpy()** is

```
char *strcpy(char *str1, const char *str2);
```

The function copies all the characters in the string pointed to by **str2** into the array pointed to by **str1**. The copy operation includes the null character in **str2**. **str1** acts as the target string while **str2** is the source string. The order of the strings is similar to the assignment statement where the target string is on the left-hand side. The **strcpy()** function returns the value of **str1**.

Program 10.10 illustrates the use of `strcpy()`. It is important to note that the target array must have enough space to hold the string. The declaration

```
char target[40];
```

is used instead of

```
char *target;
```

It is because the second declaration does not have space allocated to hold the string.

Program 10.10 Copying strings.

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

main(void)
{
    char target[40] = "This is the target string.";
    char *source = "This is the source string.";

    printf("Before strcpy():\n");
    printf("puts(target): ");
    puts(target);
    printf("puts(source): ");
    puts(source);
    strcpy(target, source);
    printf("After strcpy():\n");
    printf("puts(target): ");
    puts(target);
    printf("puts(source): ");
    puts(source);
    return 0;
}
```

Program output

```
Before strcpy():
puts(target): This is the target string.
puts(source): This is the source string.
After strcpy():
puts(target): This is the source string.
puts(source): This is the source string.
```

The **strncpy()** function is similar to **strcpy()**. The function prototype is

```
char *strncpy(char *str1, const char *str2, int n);
```

However, it takes a third argument **n**, an integer, which indicates the number of characters of the second string that are copied to the first string. If **n** is larger than the number of characters in the second string **str2**, the **strncpy()** function will add null characters to fill the first string **str1** after copying. However, if **n** is less than the number of characters in the second string, the function will not put a null character to terminate the first string. Thus, if the number of characters to be copied is less than the number of characters in the second string, the programmer should add a null character to terminate the resulting string.

The **strchr()**, **strrchr()** and **strstr()** Functions

The **strchr()** function searches the first occurrence (leftmost) of the character in the string, and returns a pointer to that occurrence. A null pointer will be returned if it does not find the character. The **strrchr()** function works similarly to **strchr()** except that it searches for the last occurrence (rightmost) of the character in the string, and returns a pointer to that occurrence. Similarly, a null pointer will be returned if it does not find the character in the string. The **strstr()** function searches for a substring.

Program 10.11 shows an example of using the string searching functions. The **strchr()** function returns the address of the leftmost 'e' in the string "Welcome to Problem Solving with ANSI C", while the **strrchr()** function returns the rightmost 'e' in the string. In the **strstr()** function, it returns the address of the first character 'c' in the string as this is the beginning of the substring "come to" that appears in the string.

Program 10.11 Searching in strings.

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

main(void)
{
    char str[] = "Welcome to Problem Solving with ANSI C";

    printf("The strchr result: %s\n", strchr(str, 'e'));
    printf("The strrchr result: %s\n", strrchr(str, 'e'));
    printf("The strstr result: %s\n", strstr(str, "come
        to"));
    return 0;
}
```

}

Program output

```
The strchr result: elcome to Problem Solving with ANSI C
The strrchr result: em Solving with ANSI C
The strstr result: come to Problem Solving with ANSI C
```

10.6 The ctype.h Character Functions

C also contains the character processing library, whose functions are declared in the *ctype.h* header file. These functions are used to test the nature of a character, and return true if the condition being tested is satisfied, or false otherwise. To use these functions, we must include the *ctype.h* header file in our programs. Some of the most commonly used functions are given in Table 10.3.

Table 10.3 Character testing functions in *ctype.h*.

Name	Returns True if Argument is
isalnum()	alphanumeric (alphabetic or numeric), i.e. 'A' - 'Z', 'a' - 'z' or '0' - '9'
isalpha()	alphabetic, i.e. 'A' - 'z' or 'a' - 'z'
iscntrl()	a control character, e.g. Control-B
isdigit()	a digit, i.e. '0' - '9'
isgraph()	any printing character other than a space, i.e. ASCII 33 – 127
islower()	a lowercase character, i.e. 'a' - 'z'
isprint()	a printing character, i.e. ASCII 32 – 127
ispunct()	a punctuation character
isspace()	a whitespace character, e.g. space, newline, formfeed, carriage return, i.e. ASCII 9 - 13 or 32
isupper()	an uppercase character, i.e. 'A' - 'Z'
isxdigit()	a hexadecimal digit character, i.e. '0' - '9', 'A' - 'F', 'a' - 'f'

The character testing functions are very useful. For example, when an input might contain any sequence of input characters, we can use the function such as **islower()**, **isupper()**, **isdigit()**, **isalpha()**, **isalnum()** or **isspace()** to test each input character and then process the character accordingly.

In addition to the functions that test characters in *ctype.h*, there are several character conversion functions for converting characters. The two most commonly used are `toupper()` and `tolower()`. The function `toupper()` converts lowercase characters to uppercase, while the function `tolower()` converts uppercase characters to lowercase. These two functions are commonly used to test character input, and convert all of them into either lowercase or uppercase, so that the program is not sensitive to the case of the letters the user enters. Program 10.12 shows the use of functions `isupper()`, `islower()`, `tolower()` and `toupper()`. If a character in the input string is tested to be in uppercase, it will be converted into lowercase using the function `tolower()`. Similarly, if a character in the input string is tested to be in lowercase, it will be converted to uppercase using the function `toupper()`.

Program 10.12 Using *ctype.h* character functions.

```
#include <stdio.h>
#include <ctype.h>
void convert(char *);

main(void)
{
    char str[80];

    printf("Enter a string of text: \n");
    gets(str);
    convert(str);
    puts(str);
    return 0;
}

void convert(char *s)
{
    while (*s != '\0')
    {
        if (isupper(*s))
            *s = tolower(*s);
        else if (islower(*s))
            *s = toupper(*s);
        s++;
    }
}
```

Program input and output

```
Enter a string of text:
This is a test
```

```
tHIS IS A TEST
```

10.7 String to Number Conversions

There are two ways to store a number. It can be stored as strings or in numeric form. For example, the number 123 can be stored as a string consisting of '1', '2', '3' and the terminating character '\0'. Sometimes, it is convenient to read in the numerical data as a string and convert it into the numeric form. To do this, C provides three functions: **atoi()**, **atol()** and **atof()**. To use these functions, we must include the *stdlib.h* file in the program:

```
#include <stdlib.h>
```

The **atoi()** function converts a character string into an integer and returns the integer value. The function prototype is

```
int atoi(const char *ptr);
```

The **atoi()** function processes the digits in the string and stops when the first nondigit character is encountered. Leading blanks are ignored and leading algebraic sign (+/-) can be recognized. The **atol()** function converts a string to a long integer value, and the **atof()** function converts a string into a double precision floating point value. Program 10.13 illustrates the **atof()** function for string to number conversion.

Program 10.13 Converting string to number.

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

main(void)
{
    char number[80];
    int i;
    double f;

    printf("Enter your number: ");
    scanf("%s", number);
    for (i=0; isdigit(number[i]); i++);
    if (number[i] != '\0')
        printf("The input is not a number.\n");
    else {
```

```
    f = atof(number);  
    printf("Input is %f\n", f);  
}  
return 0;  
}
```

Program output

```
Enter your number: 159  
Input is 159.000000
```

10.8 Formatted String Input and Output

The C standard I/O library provides two functions for performing formatted input and output to strings. These functions are **sscanf()** and **sprintf()**. The function **sscanf()** is similar to **scanf()**. The only difference is that **sscanf()** takes input characters from a string instead of from the standard input, i.e. the keyboard. It reads characters from a string and converts them into data of different types, and stores them into variables. The **sscanf()** function can be used to transform numbers represented in characters or strings (e.g. "123") into numeric numbers (e.g. 123, 123.0) of data types **int**, **float**, **double**, etc. The syntax for the function **sscanf()** is

```
sscanf(string_ptr, control-string, argument-list);
```

where **string_ptr** is a pointer to a string containing the characters to be processed. The other arguments of **sscanf()** are the same as those in **scanf()**.

The function **sprintf()** is similar to **printf()**. The only difference is that **sprintf()** prints output to a string. It formats the input data and stores them in a string. It appends a null character at the end of the string. The **sprintf()** function can be used to combine several elements into a single string. It can also be used to transform numbers into strings. The syntax for the function **sprintf()** is

```
sprintf(string_ptr, control-string, argument-list);
```

where **string_ptr** is a pointer to a string. The other arguments are the same as those in **printf()**.

Program 10.14 Formatting string input and output.

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

```
main(void)
{
    char str1[80] = "string 1369 531";
    char str2[80], str3[80];
    int i, j;

    sscanf(str1, "%s %d %d", str2, &i, &j);
    sprintf(str3, "%d %d %s", j, i, str2);
    printf("%s\n", str3);
    return 0;
}
```

Program output

```
531 1369 string
```

Program 10.14 illustrates the use of the `sscanf()` and `sprintf()` functions. The `sscanf()` function is useful to convert numbers in a string to its corresponding numeric value. The `sprintf()` function is useful to combine different data items into a string. In Program 10.14, the function `sscanf()` reads from the string `str1` and stores the elements of the string into another string `str2`, and two integers, `i` and `j`. The function `sprintf()` then combines the three elements in a different order and stores them in a new string `str3`. Figure 10.8 illustrates the operation of the `sscanf()` and `sprintf()` function.

10.9 Arrays of Character Strings

It is possible to define an array of character strings. For example, we can declare

```
char *nameptr[4] = {"Peter", "John", "Vincent", "Kenny"};
```

where `nameptr` is a one-dimensional array of four pointers to type `char`. Each pointer points to the first character of the corresponding string. That is, the first pointer `nameptr[0]` points to the first character of the first string, the second pointer `nameptr[1]` points to the first character of the second string, and so on. `nameptr` is an array of pointers. This is also called a *ragged array*.

We can then use an index (or subscript) to access each element of the array for the character strings as follows:

```
for (i=0; i<4; i++)
    printf("nameptr[%d] = %s\n", i, nameptr[i]);
```

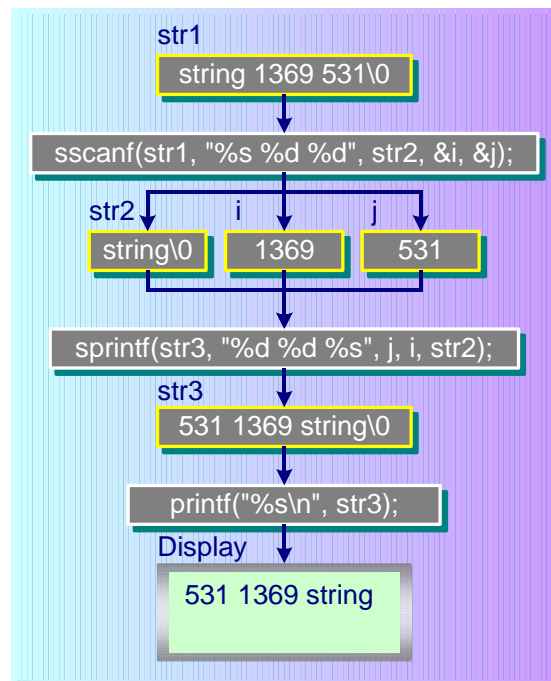


Figure 10.8 The `sscanf()` and `sprintf()` functions.

To print a character from the string, we can use the indirection operator (i.e. '*'). For example, the statement

```
printf("%c", *(nameptr[0] + 2));
```

prints the character 't' from the string "Peter".

Another way to store array of strings is to use a two-dimensional array. For example, we can declare **name** as

```
char name[4][8] = {"Peter", "John", "Vincent", "Kenny"};
```

This is called a *rectangular array*. All the rows are of the same length. Figure 10.9 shows the array of pointers **nameptr** and the rectangular array **name**. The array of pointers defined in **nameptr** helps to save storage space. The two-dimensional array **name** needs to be defined with enough storage space to hold the longest string, thus some space is wasted as not all the other strings will have the same length as the longest string. For array of pointers, no space will be wasted. Program 10.15 illustrates the processing of the ragged array and rectangular array.

Program 10.15 Processing arrays of character strings.

```
#include <stdio.h>

main(void)
{
    char *nameptr[4] = {"Peter", "John", "Vincent", "Kenny"};
    char name[4][8] = {"Peter", "John", "Vincent", "Kenny"};
    int i, j;

    printf("Ragged Array: \n");
    for (i=0; i<4; i++)
        printf("nameptr[%d] = %s\n", i, nameptr[i]);
    printf("\nRectangular Array: \n");
    for (j=0; j<4; j++)
        printf("name[%d] = %s\n", j, name[j]);
    return 0;
}
```

Program output

```
Ragged Array:
ptr_name[0] = Peter
ptr_name[1] = John
ptr_name[2] = Vincent
ptr_name[3] = Kenny

Rectangular Array:
name[0] = Peter
name[1] = John
name[2] = Vincent
name[3] = Kenny
```

10.10 Command Line Arguments

The command line is the string of characters we type in order to run a program. Arguments can be given to commands as options. For example, in the commands

```
ls -las
cat file1 file2 file3
```

where **-las** is the argument for **ls**, and **file1**, **file2** and **file3** are the arguments for **cat**. Users can also supply arguments to their application programs.

The `main()` function receives arguments when a C program is executed. The syntax to receive these arguments is

```
main(int argc, char *argv[])
{
    ....
}
```

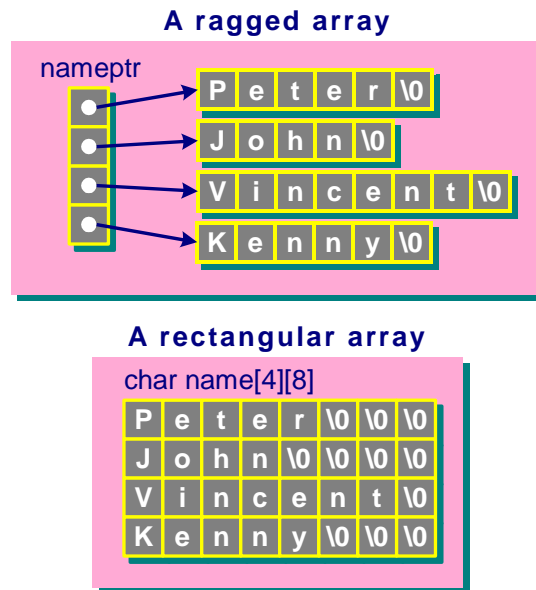


Figure 10.9 Array of pointers and rectangular array.

where `argc` is the argument counter which reports how many words are in the command line. This includes the command itself, e.g. `a.out`, `ls`, `cat`, etc., and `argv` is the argument value represented by an array of pointers pointing to the input strings. The number of pointers in `argv` is equal to `argc`. For example, if the name of the executable program is `prog`, and the command line

```
prog command line arguments
```

gives the value for `argc` as 4, where `argv[0]` points to "`prog`", `argv[1]` points to "`command`", `argv[2]` points to "`line`", and `argv[3]` points to "`arguments`". This is illustrated in Figure 10.10. The C program is shown in Program 10.16. Command line arguments are commonly used to pass file names and other parameters to a program.

Program 10.16 Using command line arguments.

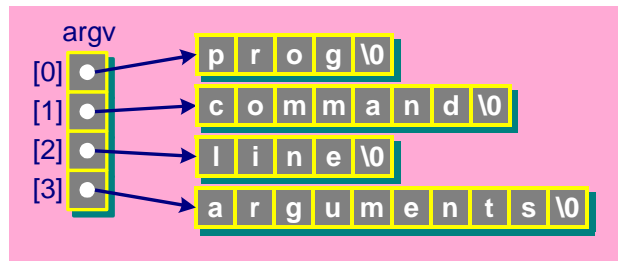
```
#include <stdio.h>

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

    printf("Command Line Arguments:\n");
    printf("argc = %d \n", argc);
    for (count = 0; count < argc; count++)
        printf("argv[%d] = %s \n", count, argv[count]);
    return 0;
}
```

Program input and output

```
$ prog command line arguments
Command Line Arguments:
argc = 4
argv[0] = prog
argv[1] = command
argv[2] = line
argv[3] = arguments
```

**Figure 10.10** Command line arguments.**10.11 Exercises**

1. What is the relationship between an *array* and a *string*?
2. Explain the difference between the following declarations:

```
char str1[ ] = { 'a', 'b', 'c', 'd' };
char str2[ ] = "Hello ?";
char *str3 = "How are you ?";
```

State whether the following statements are valid. Explain your answers.

- (a) `str2 = "I am fine."`
- (b) `str3 = "I am fine."`

3. The following `unknown()` function receives a string argument and a character argument, modifies the string argument and returns an integer value. Describe the purpose of the function. Give an example to support your answer.

```
int unknown(char str[ ], char c)
{
    int x, y=0, z=0;

    for (x=0; str[x] != '\0'; x++)
        if (str[x] != c)
            str[y++] = str[x];
        else
            z++;
    str[y] = '\0';
    return z;
}
```

4. Use the input `STAR` to trace the working of the following `unknown()` function.

```
void unknown(char *str)
{
    char temp, *end;
    end = str + strlen (str) - 1;
    while(str < end) {
        temp = *str;
        *str++ = *end;
        *end-- = temp;
    }
}
```

5. Consider the following `unknown()` function. State the purpose of the function. Use the string `"How are you?"` as the input to trace the working of the function, and determine the output of the function.

```
void unknown (char *str1, char *str2)
{
```

```

int i, isspace = 1;

for (i=0; i < strlen(str1); i++)
{
    if (isspace == 1 && str1[i] != ' ') {
        *str2 = str1[i];
        str2++;
        isspace = 0;
    }
    if (str1[i] == ' ')
        isspace = 1;
}
*str2 = '\\0';
}

```

6. The function `locatechr()` has the following function prototype:

```
char *locatechr(char *str, char ch);
```

This C function locates the last occurrence of `ch` in the string pointed to by `str`. The function returns a pointer to the character, or a null pointer if `ch` does not occur in the string.

7. Write a C program `add` that adds two numbers of integer values and prints the result on the display. The numbers are specified as command line arguments. For example, the following command

```
add 111 222
```

can be activated to add the two numbers 111 and 222, and it will then display the result 333.

8. Determine the outputs of the following program.

```

#include <stdio.h>
main( )
{
    char a[3][5] = { { 'a', 'b', 'c', 'd', 'e' },
                     { 'f', 'g', 'h', 'i' },
                     { 'j', 'k', 'l' }
                   };
    char *b = "0123456789";
    char *c[2];
    char *ptr;

```

```
    ptr = &a[1][1];
    c[0] = &a[0][0];
    c[1] = &a[1][0];
    printf ("%a[1][2] = %s\n", &a[1][2]);
    printf ("%a[1][4] = %s\n", &a[1][4]);
    printf ("%b[4] = %s\n", &b[4]);
    printf ("c[0]+1 = %s\n", c[0]+1 );
    printf ("c[1]+2 = %s\n", c[1]+2 );
    printf ("ptr-3 = %s\n", ptr-3);
    return 0;
}
```



Structures, Unions and Enumerated Types

In Chapter 9, we introduced arrays, which store a collection of unrelated data items of the same data type. C also provides a data type called *structure* that stores a collection of data items of different data types as a group. The individual components of a structure can be any valid data types. In this chapter, we describe **struct**, **union** and **enum** data types.

This chapter covers the following topics:

- 11.1 Structures
- 11.2 Arrays of Structures
- 11.3 Nested Structures
- 11.4 Pointers to Structures
- 11.5 Functions and Structures
- 11.6 The **typedef** Construct
- 11.7 Unions
- 11.8 Enumerated Data Types

11.1 Structures

Structure is an aggregate of values. Their components are distinct, and may possibly have different types, including arrays and other structures. To build our own data types using structures, we need to define the structure and declare variables of that type. A structure template is used to specify a structure definition.

It tells the compiler the various components that make up the structure. Structure variables are then declared with the type of the structure.

Structure Template

A structure template is the master plan that describes how a structure is put together. A structure template can be set up as follows:

```
struct book {           /* template of book*/
    char title[40];     /* members of the structure */
    char author[20];
    float value;
    int libcode;
};                      /* semicolon to end the definition */
```

struct is a reserved keyword to introduce a structure. **book** is an optional tag name that follows the keyword **struct** to name the structure declared. The **title**, **author**, **value** and **libcode** are the *members* of the structure **book**. The members of a structure can be any of the valid C data types. A semicolon after the closing brace ends the definition of the structure definition. The above declaration declares a template, not a variable. Therefore, no memory space is allocated. It only acts as a template for the named structure type. The tag name can then be used in the declaration of variables. Program 11.1 illustrates an example on defining structure template and the declaration of structure variable.

Program 11.1 Structure template and structure variables.

```
#include <stdio.h>
struct book {
    char title[40];
    char author[20];
    float value;
    int libcode;
};

int main(void)
{
    struct book bookRec;

    printf("Please enter the book title: \n");
    gets(bookRec.title);
    printf("Please enter the author: \n");
    gets(bookRec.author);
    printf("Please enter the value: \n");
    scanf("%f", &bookRec.value);
```

```

printf("Please enter the library code: \n");
scanf("%d", &bookRec.libcode);
printf("The book %s (%d) by %s: $%.2f.\n", bookRec.title,
      bookRec.libcode, bookRec.author, bookRec.value);
return 0;
}

```

Program input and output

```

Please enter the book title:
Introduction to C Programming
Please enter the author:
SC Hui
Please enter the value:
25.00
Please enter the library code:
123456
The book Introduction to C Programming (123456) by SC Hui:
$25.00.

```

The declaration

```
struct book bookRec;
```

introduces a variable **bookRec** of type **book**. It also allocates storage for the variable. The structure definition can be placed inside a function or outside a function. If it is defined inside the function, the definition can only be used by that function. In the program, the definition is defined at the beginning of the file. It is a global declaration, and all the functions following the definition can use the template. Figure 11.1 shows the storage for the structure variable **bookRec**.

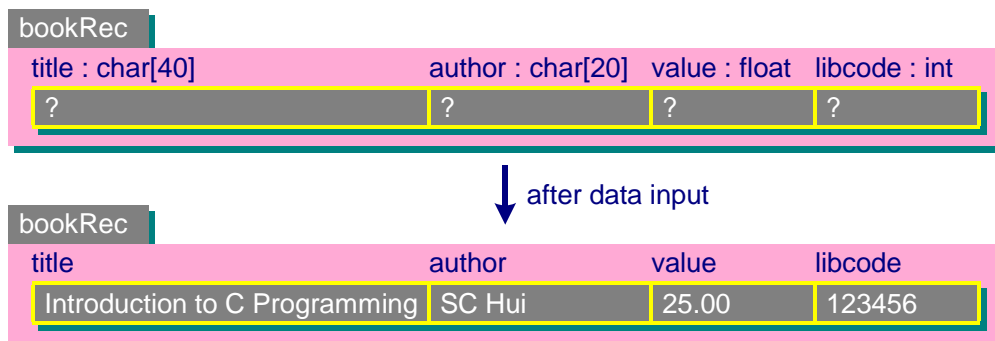


Figure 11.1 Storage for the structure variable **bookRec**.

Structure Tags and Structure Variables

The structure name or tag is optional. With structure tag, the definition of structure template can be separated from the definition of structure variables. In the following declaration, a structure template **person** comprising three components is created. **tom** and **mary** are two structure variables which are declared using the structure **person**.

```
struct person {           /* with tag */
    char name[20];
    int age;
    float salary;
};
struct person tom, mary; /* structure variables */
```

Without structure tag, the definition of structure template must be combined with that of structure variables. In the following declaration, a structure template is created with three components. The variables **tom** and **mary** are then defined using this structure. Figure 11.2 shows the memory storage for the two structure variables **tom** and **mary**.

```
struct {                  /* no tag */
    char name[20];
    int age;
    float salary;
} tom, mary;              /* structure variables */
```

Without structure tag name, we cannot use the structure elsewhere in the program. It is always a good idea to include structure tag when defining a structure.

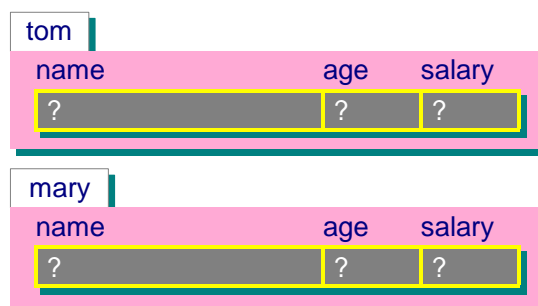


Figure 11.2 Defining two structure variables without initialization.

Structure Initialization

The syntax for initializing structures is similar to that of initializing arrays. When there are insufficient values to be assigned to all members of the structure, the remaining members are assigned to zero by default. The structure variable is followed by an assignment symbol and a list of values defined within braces:

```
struct personTag {           /* with tag */
    char  name[40];
    char  id[20];
    char  tel[20];
} student = {"John", "CE000011", "123-4567"};
                                   /* with initialization */
```

Initialization of variables can only be performed with constant values or constant expressions that deliver a value of the required type. The initial values are assigned to the individual members of the structure in the order in which the members occur. The **name** member of **student** is assigned with "John", the **id** member is assigned with a value of "CE000011", and the **tel** member is assigned with "123-4567". This is illustrated in Figure 11.3.

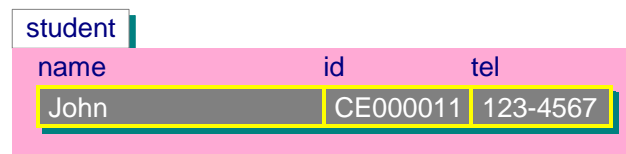


Figure 11.3 Defining a structure variable with initialization.

Accessing Structure Members

The notation required to access a member of a structure is

```
structureVariableName.memberName
```

For example, to access the member **id** of the variable **student**, we use

```
student.id
```

The "." is an access operator known as the *member operator*. The member operator has the highest (or equal) priority among the operators.

Structure Assignment

The value of one structure variable can be assigned to another structure variable of the same type using the assignment operator:

```
struct personTag newmember;  
newmember = student;
```

This has the effect of copying the entire contents of the structure variable **student** to the structure variable **newmember**. Each member of the **newmember** variable is assigned with the value of the corresponding member in the **student** variable.

11.2 Arrays of Structures

A structure variable can be seen as a record. For example, the structure variable **student** in the previous section is a student record with the information of a student name, identity and telephone number. When structure variables of the same type are grouped together, we have a database of that structure type. One can create a database by defining an array of certain structure type as shown in Figure 11.4.

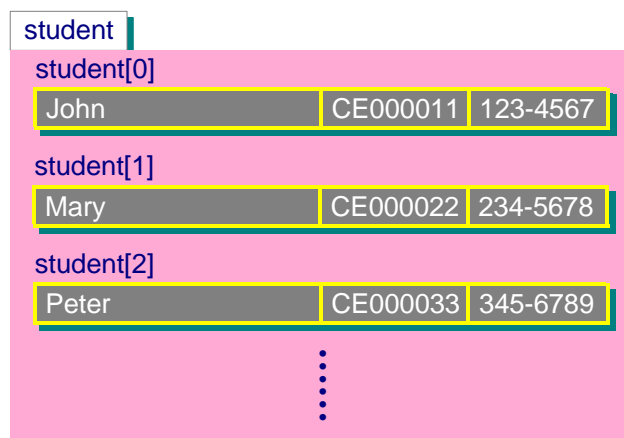


Figure 11.4 An array of structures as a database.

Program 11.2 shows an example on array of structures. The variable **student** defines an array of structures, which is a database of student records. The array has three elements and each element of the array is a structure of type **personTag**. This means that each element of the array has three members, namely **name**, **id**

and **tel**. The syntax for declaring an array of structures is

```
struct personTag student[3];
```

where it starts with the keyword **struct**, and followed by the name of the structure **personTag** that identifies the data type. This is then followed by the name of the array, **student**. The values specified within the square brackets specify the total number of elements in the array.

Array index is used when accessing individual elements of an array of structures.

```
student[i]
```

denotes the (i+1)th record. The first element starts with index 0. To access a member of a specific element, we use

```
student[i].name
```

which denotes a member of the (i+1)th record. We use

```
student[i].name[j]
```

to denote a single character value of a member of the (i+1)th record.

Array of structures can be initialized. The initializers for each element are enclosed in braces, and each member is separated by a comma. An example is given as follows:

```
struct personTag student[3] = {
    { "John", "CE000011", "123-4567"},
    /* initialized values for student[0] */
    { "Mary", "CE000022", "234-5678"},
    /* initialized values for student[1] */
    { "Peter", "CE000033", "345-6789"},
    /* initialized values for student[2] */
};
```

Program 11.2 Array of structures to form a database.

```
#include <stdio.h>
struct personTag {
    char name[40],id[20],tel[20];
};
struct personTag student[3] = {
    { "John", "CE000011", "123-4567"},
```

```

    {"Mary", "CE000022", "234-5678"},
    {"Peter", "CE000033", "345-6789"},
};

main(void)
{
    int i;
    for (i=0; i<3; i++)
        printf("Name: %s, ID: %s, Tel: %s\n", student[i].name,
            student[i].id, student[i].tel);
    return 0;
}

```

11.3 Nested Structures

A structure can also be included in other structures. This is called nested structures. For example, to keep track of the course history of a student, one can use a structure (without any nested structures) as follows:

```

struct studentTag {
    char  name[40];
    char  id[20];
    char  tel[20];
    int   SC101Yr;    /* the year when SC101 is taken */
    int   SC101Sr;    /* the semester when SC101 is taken */
    char  SC101Grade; /* the grade obtained for SC101 */
    int   SC102Yr;    /* the year when SC102 is taken */
    int   SC102Sr;    /* the semester when SC102 is taken */
    char  SC102Grade; /* the grade obtained for SC102 */
};
struct studentTag student[1000];

```

Alternatively, the variable `student` can be defined in a more elegant manner using nested structures as follows:

```

struct personTag {
    char  name[40];
    char  id[20];
    char  tel[20];
};
struct courseTag {
    int   year, semester;
    char  grade;
};

```

```

struct studentTag {
    struct personTag  studentInfo;
    struct courseTag  SC101, SC102;
};
struct studentTag student[1000];

```

In the nested structure declaration, the first structure creates a structure template called **personTag** that has three members, **name**, **id** and **tel**, of the array data type. The second definition creates a structure template called **courseTag**. The structure **courseTag** has three members, namely, **year** and **semester** of type **int**, and **grade** of type **char**. The third structure definition has three members. The member **studentInfo** is a structure of **personTag**, while the other two members, **SC101** and **SC102**, are structure of **courseTag**. Notice that the structure definition of **personTag** and **courseTag** must appear before the definition of structure **studentTag**. In summary, we have

- **student**, which denotes the complete array (i.e. the database);
- **student[i]**, which denotes the (i+1)th record;
- **student[i].studentInfo**, which denotes the personal information in the (i+1)th record;
- **student[i].studentInfo.name**, which denotes the student name in the (i+1)th record; and
- **student[i].studentInfo.name[j]**, which denotes a single character value in the (i+1)th record.

For example, consider the following nested structure declaration with initialization:

```

struct studentTag newstudent[3] = {
    {"John", "CE000011", "123-4567"},
        {2002, 1, 'B'}, {2002, 1, 'A'}},
    {"Mary", "CE000022", "234-5678"},
        {2002, 1, 'C'}, {2002, 1, 'A'}},
    {"Peter", "CE000033", "345-6789"},
        {2002, 1, 'B'}, {2002, 1, 'A'}}
};

```

The following statements print the values of members of individual elements of the **newstudent** array:

```

int i;
for (i=0; i<2; i++) {
    printf("Name:%s, ID: %s, Tel: %s\n",
        student[i].studentInfo.name,

```

```

        student[i].studentInfo.id,
        student[i].studentInfo.tel);
printf("SC101 in year %d semester %d : %c\n",
        student[i].SC101.year,
        student[i].SC101.semester,
        student[i].SC101.grade);
printf("SC102 in year %d semester %d : %c\n",
        student[i].SC102.year,
        student[i].SC102.semester,
        student[i].SC102.grade);
}

```

11.4 Pointers to Structures

Pointers are flexible and powerful in C. They can be used to point to structures. The declarations

```

struct personTag {
    char name[40],id[20],tel[20];
};
struct personTag student={"John","CE000011","123-4567"};
struct personTag *ptr;

```

define a structure template **personTag** and create a pointer **ptr** to the structure **personTag**. To initialize a pointer, we must use the address operator (**&**) to obtain the address of a structure variable, and then assign the address to the pointer as

```
ptr = &student;
```

The address of the structure variable **student** is assigned to the pointer variable **ptr**. Figure 11.5 shows the use of a pointer to access structure members.

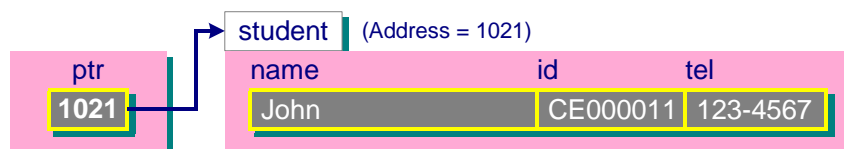


Figure 11.5 Using a pointer to access structure members.

The indirection operator (*****) can be used to access a member of a structure via a pointer to the structure. Since **ptr** points to **student**, the notations

```
(*ptr).name, (*ptr).id, (*ptr).tel
```

return the value of the member **name**, **id** and **tel** of the **student**. The parentheses are necessary to enclose ***ptr** as the member operator (.) has higher precedence than the indirection operator (*).

Since dereferencing is very common in pointer to structure, C provides an operator called the *structure pointer operator* (->) for a pointer pointing to a structure. There is no whitespace between the symbols (-) and (>). We can use the notations

```
ptr->name, ptr->id, ptr->tel
```

to obtain the values of the members of the structure **student**. It takes less typing if **ptr->tel** is compared with **(*ptr).tel**, though they have exactly the same meaning. It is quite common to use structure pointer operator (->) instead of the indirection (*) operator in pointers to structures.

There are three reasons of using pointers to structures:

1. Pointers to structures are easier to manipulate than structures themselves.
2. In older C implementation, structure is passed as an argument to a function using pointers to structures.
3. Many advanced data structures require pointers to structures.

Program 11.3 illustrates pointers to structures.

Program 11.3 Pointers to structures.

```
#include <stdio.h>
struct book {
    char title[40];
    char author[20];
    float value;
    int libcode;
};

main(void)
{
    struct book bookRec = {
        "Introduction to C Programming",
        "SC Hui",
        25.00,
        123456 };
    struct book *ptr;

    ptr = &bookRec;
```

```
printf("The book %s (%d) by %s: $%.2f.\n", ptr->title,  
      ptr->libcode, ptr->author, ptr->value);  
return 0;  
}
```

Program output

```
The book Introduction to C Programming (123456) by SC Hui:  
$25.00.
```

11.5 Functions and Structures

Older C implementations do not allow a structure to be used as an argument for a function. However, ANSI C allows structures to be used as arguments. For example, the function with the following function prototype is allowed.

```
void printStudentRec(struct personTag);
```

It is often necessary to pass structure information to a function. In C, there are four ways to pass structure information to a function:

1. Pass structure members as arguments using call by value, or call by reference;
2. Pass structures as arguments;
3. Pass pointers to structures as arguments; and
4. Pass by returning structures.

Passing Structure Members as Arguments

Individual members of a structure can be passed as arguments to a function. This is shown in Program 11.4.

Program 11.4 Passing structure members as arguments.

```
#include <stdio.h>  
float sum(float, float);  
struct account {  
    char    bank[20];  
    float   current;  
    float   saving;  
};  
  
main(void)  
{
```



```
    struct account john = {"OCBC Bank", 1000.43, 4000.87};  
    printf("The account has a total of %.2f.\n",  
        sum(john.current, john.saving));  
    return 0;  
}  
float sum(float x, float y)  
{  
    return (x+y);  
}
```

Program output

The account has a total of 5001.30.

The function `sum()` expects two arguments, `x` and `y`, of type `float`. The structure variable `john` is declared with `struct account` and the values of the members `current` and `saving` are passed to the function `sum()`. The structure members `john.current` and `john.saving` are of type `float`. As long as a structure member is a variable of a data type with a single value, we can pass the structure member as a function argument. The structure members `john.current` and `john.saving` are passed by value. It means that local copies of the variables are created in the function `sum()`. Any changes made to the function will only affect the local copies of the variables.

It is also possible to pass the structure members using call by reference. However, we need to pass the address of the members to the function `sum()`:

```
void sum(float *x, float *y, float *result)  
{  
    *result = *x + *y;  
}
```

This new function receives the addresses of the variables `x` and `y` of type `float`. The sum of the content of the memory locations pointed to by the two variables is then calculated and stored in the memory location pointed to by the pointer variable `result`. The function call using the address will be

```
sum(&john.current, &john.saving, &sum_value);
```

where the variables `john` and `sum_value` of types `struct account` and `float` are defined respectively in the calling function.

Passing Structures as Arguments

Another way to pass a structure information to a function is to pass the entire structure as an argument to a function. ANSI C standard supports this feature. An example is given in Program 11.5.

Program 11.5 Passing a structure as an argument.

```
#include <stdio.h>
float sum(struct account); /* argument is a structure */
struct account {
    char    bank[20];
    float   current;
    float   saving;
};

main(void)
{
    struct account john = {"OCBC Bank", 1000.43, 4000.87};
    printf("The account has a total of %.2f.\n", sum(john));
    return 0;
}

float sum(struct account money)
{
    return(money.current + money.saving);
    /* not money->current */
}
```

Program output

```
The account has a total of 5001.30.
```

When a structure is passed as an argument to a function, it is passed using call by value. The members of this structure in the function `sum()` are initialized with local copies. The function can only modify the local copies, and cannot change the original variables. Notice that we simply use the member operator (`.`) to access the individual members of the structure variable as follows:

```
return(money.current + money.saving);
```

Figure 11.6 illustrates the concept on passing a structure to a function. The advantage of using this method is that the function cannot modify the members of the original structure variables, which is safer than working with the original variables. However, this method is quite inefficient to pass large structures to

functions. In addition, it also takes time and additional storage to make a local copy of the structure. Moreover, some older C compilers do not support this approach.

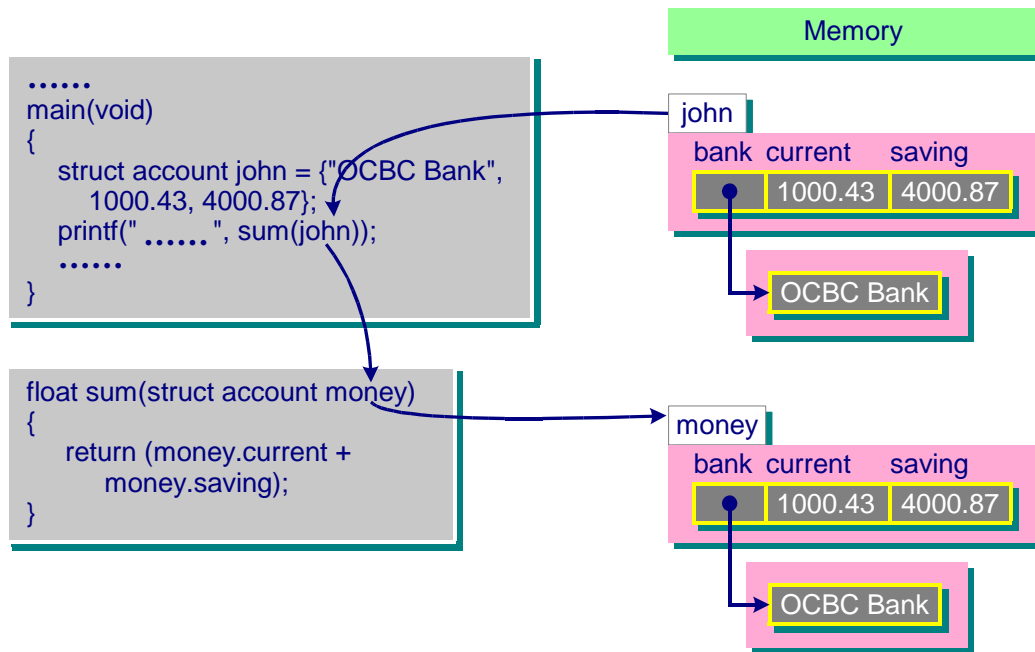


Figure 11.6 Illustration of program 11.5.

Passing Structure Address

We can use the address of the structure as an argument. Using the same structure template `account`, Program 11.6 shows the use of the structure address as an argument.

Program 11.6 Passing structure address to a function.

```
#include <stdio.h>
float sum(struct account *);    /* argument is a pointer */
struct account {
    char    bank[20];
    float   current;
    float   saving;
};

main(void)
{
    struct account john = {"OCBC Bank", 1000.43, 4000.87};
```

```

    printf("The account has a total of %.2f.\n", sum(&john));
    return 0;
}
float sum(struct account *money)
{
    return(money->current + money->saving);
}

```

Program output

The account has a total of 5001.30.

The `sum()` function uses a pointer to an account structure as its argument. The address of `john` is passed to the function that causes the pointer `money` to point to the structure `john`. The `->` operator is then used in the following statement:

```
return(money->current + money->saving);
```

to obtain the values of `john.current` and `john.saving`. This allows the function to access the structure variable and to modify its content. This is a better approach than passing structures as arguments. Figure 11.7 illustrates the concept on passing a pointer to structure to a function.

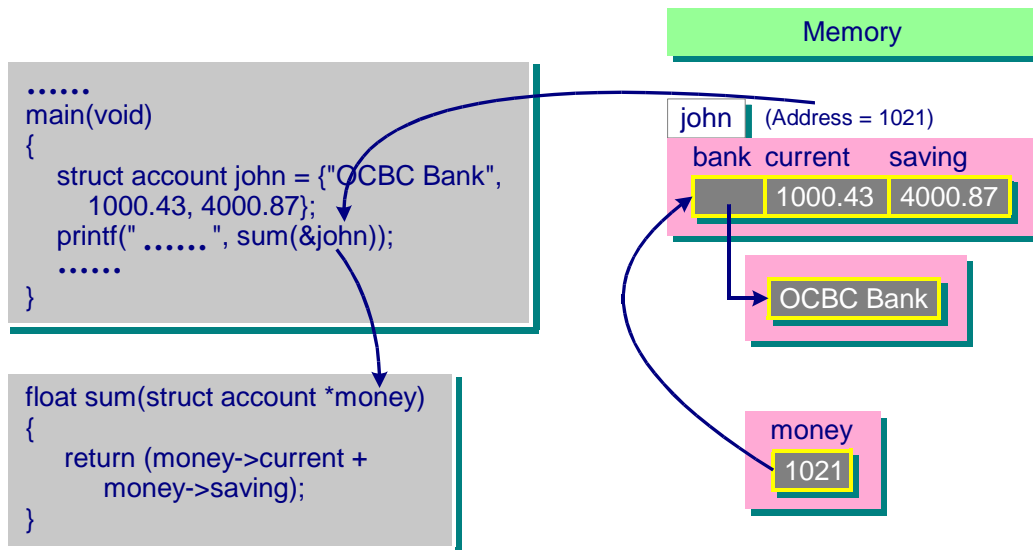


Figure 11.7 Illustration of Program 11.6.

Passing by Returning Structures

Functions can return structures. This is illustrated in Program 11.7. The function

```
struct nameTag getname(void);
```

returns a structure `nameTag`. To call this function, the calling function must declare a variable of type `struct nameTag` in order to receive the result from `getname()`:

```
struct nameTag name;  
name = getname();
```

Program 11.7 Passing by returning structures.

```
#include <stdio.h>  
struct nameTag {  
    char fname[20];  
    char lname[20];  
};  
struct nameTag getname(void);  
  
main(void)  
{  
    struct nameTag name;  
    name = getname();  
    printf("Your name is %s %s\n", name.fname, name.lname);  
    return 0;  
}  
struct nameTag getname(void)  
{  
    struct nameTag newname;  
    printf("Please enter the first name:\n");  
    gets(newname.fname);  
    printf("Please enter the last name:\n");  
    gets(newname.lname);  
    return newname;  
}
```

Program input and output

```
Please enter the first name:  
Siu Cheung  
Please enter the last name:  
Hui  
Your name is Siu Cheung Hui
```

11.6 The typedef Construct

typedef provides an elegant way in structure declaration. The general syntax for the **typedef** statement is

```
typedef datatype UserProvidedName;
```

The **typedef** keyword is followed by the data type and the user provided name for the data type. It is very useful for creating simple names for complex structures. For example, if we have defined the structure:

```
struct date {  
    int day, month, year;  
};
```

we can define a new data type **Date** as

```
typedef struct date    Date;
```

Variables can then be defined either as

```
    struct date    today, yesterday;  
or    Date        today, yesterday;
```

We can also use the type **Date** in function prototypes and function definitions. When **typedef** is used, tag name is redundant. Therefore, we can declare

```
typedef struct {  
    int day, month, year;  
} Date;  
Date today, yesterday;
```

There are a number of advantages of using **typedef** statements. It enhances program documentation by using meaningful names for data types in the programs. It makes the program easier to read and understand. Another advantage is to define simpler data types for complex declarations such as structures. Moreover, **typedef** can make programs more portable. For instance, suppose our program needs to use 32-bits (4-bytes) integer values. Since some systems use 2-bytes for **int** and some use 4-bytes for **int**, we need to use the type **long** instead for those systems implementing 2-bytes for **int**. Therefore, if only type **long** or **int** is used in the declarations, we need to change code when moving from one system to another. To tackle this, we can use the **typedef** statement for systems with 4-bytes integers:

```
typedef int FOURBYTE_INT;
```

For other systems that type `long` is needed, we can use

```
typedef long FOURBYTE_INT;
```

It is convenient to place the `typedef` statements in a separate header file.

In addition, `typedef` is similar to the `#define` preprocessor directive. However, there are a number of differences. `typedef` is limited to giving names to data types only and is processed by the compiler, while `#define` is not limited to data types and is processed by the preprocessor.

11.7 Unions

Union is similar to structure in that union also contains members of different types and sizes. However, union can hold at most one of its members at a time. Members are overlaid in the storage allocated for the union. Compiler allocates sufficient storage to accommodate the largest of the union members.

The syntax for creating and using unions is the same as for structures. A `union` template can be defined with an optional tag, which can then be used for creating variables. In the following declaration:

```
union data {  
    int digit;  
    double real_num;  
    char letter;  
};
```

it declares a `union` type having the tag name as `data`. This can be used to store an integer, a double or a character. Figure 11.8 illustrates the `union data`.

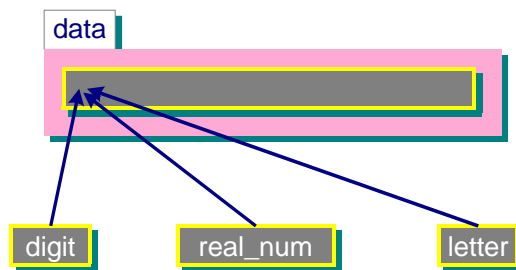


Figure 11.8 Union data structure.

We can then define a variable **data1** as

```
union data data1;
```

Members of a union can be accessed using

```
unionVariable.memberName /* using a union variable */  
or unionPointerVariable->memberName /* using a pointer */
```

To assign values to **data1**, we can use the member operator (.) as follows:

```
data1.digit = 10; /* 2 bytes are used to store 10 */  
data1.real_num = 10.0; /* 8 bytes of the same memory are  
used to store 10.0 */  
data1.letter = 'A'; /* 1 byte is used to store 'A' */
```

It is the programmer's responsibility to keep track of the data type currently being stored in a **union**.

We can use the **->** operator with pointers to unions in a similar way as pointers to structures:

```
data_ptr = &data1;  
num = data_ptr->digit;
```

This assignment statement accesses the value of the **digit** member and assigns the value to the variable **num**. After the assignment operation, **num** will contain the value 10. Therefore, we can declare and access members of unions in the same way as the structures, except that the keyword **struct** is replaced by **union** in the declaration.

A **union** can occur within a structure, or a structure can also occur within a **union**. This is further illustrated in Program 11.8. In Program 11.8, a fleet of ships can include passenger liners and cargo ships. The characteristics of the ships can be represented using the following declarations:

```
typedef struct {  
    int name[21];  
    int crew_number;  
    int seats;  
    int beds;  
} passengerClass;  
  
typedef struct {  
    int name[21];  
    int crew_number;
```



```

    int capacity;
    int volume;
} cargoClass;

```

To represent a fleet of ships, we can define a general ship type which can represent any ships as a structure template:

```

typedef struct {
    char name[21],
    int crew_number;
    int seats, beds;
    int capacity, volume;
} shipClass;

```

or alternatively we can use **union** (to save space) of structures as follows:

```

typedef union {
    passengerClass passenger;
    cargoClass cargo;
} shipClass;

```

Program 11.8 uses **union** to capture the structures of two different classes of ships, namely passenger liners and cargo ships. To create a variable **ships** of type **shipClass**, we declare

```

shipClass ships[10];

```

which creates a fleet of 10 ships. The compiler will allocate storage to hold the array of **struct union**. For each element of the array of **ships**, the compiler will allocate enough storage to hold the largest member of the **struct union**. We can use the following assignment statements to assign values to the members of an element of the array **ships**.

```

ships[0].passenger.shipType = PASSENGER;
strcpy(ships[0].passenger.name, "Washington");
ships[0].passenger.crew_number = 100;
ships[0].passenger.seats = 300;
ships[0].passenger.beds = 220;

```

Program 11.8 Nested unions and structures.

```

#include <stdio.h>
#include <string.h>
#define PASSENGER 1

```



```
        printf("\tseats = %d\n", ship.passenger.seats);
        printf("\tbeds = %d\n", ship.passenger.beds);
    }
    else {
        /* shipType == CARGO */
        printf("Cargo ship:\n");
        printf("\tname = %s\n", ship.cargo.name);
        printf("\tcrew_number = %d\n", ship.cargo.crew_number);
        printf("\tcapacity = %d\n", ship.cargo.capacity);
        printf("\tvolume = %d\n", ship.cargo.volume);
    }
}
```

Program output

```
Passenger liner:
    name = Washington
    crew_number = 100
    seats = 300
    beds = 220
Cargo ship:
    name = Rogers
    crew_number = 80
    capacity = 1000
    volume = 2000
```

11.8 Enumerated Data Type

Enumerated data type allows programmers to define symbolic names to represent integer constants. The keyword **enum** is used to declare enumerated types. It is followed by an optional tag and a list of names representing permissible values for this data type. For example, **Day** is declared as an enumerated type:

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };
```

Names inside the braces **{ }** are called *enumeration constants*. These constants are similar to the constants defined by the C preprocessor directive **#define**. The permissible values for variables of type **Day** are **SUN, MON, TUE, WED, THU, FRI** and **SAT**.

The new type name **Day** together with the keyword **enum** can be used to define variables just like other data types such as **int, float, double**, etc. A variable of type **enum Day** can have any value of the enumerated constants:

```
enum Day day1, day2, day3;
```

The enumerated type can also be declared together with variables:

```
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT} day1, day2,
        day3;
```

The enumeration tag **Day** in the above declaration is optional. The above declaration is equivalent to

```
enum {SUN, MON, TUE, WED, THU, FRI, SAT} day1, day2,
        day3;
```

The enumeration constants, e.g. **SUN**, are represented (by the C compiler) internally by a number starting from zero as follows:

```
SUN=0    MON=1    TUE=2    WED=3    THU=4    FRI=5    SAT=6
```

The internal representation of an enumeration constant can be overridden by explicit assignments such as

```
enum Day {SUN, MON=8, TUE, WED, THU=30, FRI, SAT};
```

This will cause the compiler to change the internal representation as

```
SUN=0    MON=8    TUE=9    WED=10   THU=30   FRI=31   SAT=32
```

We can use a series of **#define** preprocessor directives instead of enumerated types:

```
#define SUN 0
#define MON 1
#define TUE 2
#define WED 3
#define THU 4
#define FRI 5
#define SAT 6
```

The advantage of enumerated data type is the freedom to program using constant names, e.g. **MON**, **TUE**, etc., instead of numbers. Symbolic names are more readable than numbers. Program 11.9 illustrates an example on using enumerated types.

Program 11.9 Enumerated types.

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

```
main(void)
{
    enum Day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
    enum Day day1;

    for (day1 = Sun; day1 <= Sat; day1++) {
        switch (day1) {
            case Sun:    printf("Monday is after Sunday?\n");
                        break;
            case Mon:    printf("Tuesday is after Monday?\n");
                        break;
            case Tue:    printf("Wednesday is after Tuesday?\n");
                        break;
            case Wed:    printf("Thursday is after Wednesday?\n");
                        break;
            case Thu:    printf("Friday is after Thursday?\n");
                        break;
            case Fri:    printf("Saturday is after Friday?\n");
                        break;
            case Sat:    printf("Sunday is after Saturday?\n");
        }
    }
    return 0;
}
```

Program output

```
Monday is after Sunday?
Tuesday is after Monday?
Wednesday is after Tuesday?
Thursday is after Wednesday?
Friday is after Thursday?
Saturday is after Friday?
Sunday is after Saturday?
```

11.9 Exercises

1. Explain the differences between the data types *structure* and *union*.
2. The data type `person` and variables `person1` and `person2` have been defined as follows:

```
typedef struct {
    char  first_name[20], last_name[20];
```

```

    int    age;
    char   sex;
} person;
....
person person1, person2;

```

State whether the following statements are valid or invalid. Explain your answer if the statement is invalid.

- (a) `printf("%s", person1);`
 - (b) `person2 = person1;`
 - (c) `if (person2 == person1) printf("Identical persons");`
 - (d) `person2.lastname = "Tan";`
3. Design the data structure that captures the following book information: title of the book, author name (last name and first name), date of publication and publisher. Write a program that prompts the user to enter a book entry, and prints the entered information on the display.
4. A structure called `complex` is defined to represent a complex number. Each complex number consists of the real and imaginary parts as follows:

```

typedef struct {
    float real;
    float imag;
} complex;

```

Write C functions to perform addition, subtraction, multiplication and division operations on two complex numbers. The function prototypes are given as follows:

```

complex add_complex(complex c1, complex c2);
complex sub_complex(complex *c1, complex *c2);
complex mul_complex(complex c1, complex c2);
complex div_complex(complex *c1, complex *c2);

```

Write a C program that handles complex number arithmetic using the arithmetic functions implemented. The program will read the choice of operation (e.g. addition, subtraction, multiplication, division and quit) and two complex numbers, and display the result. Give the advantages and disadvantages of the parameter passing schemes used by the arithmetic functions.



File Input/Output

Files are used to store large amount of information on secondary storage devices such as disk storage devices. Unlike data stored as variables or arrays in programs, data stored in files are permanent which persist even when the power is shut off. The C library provides a set of functions for performing file input/output. In this chapter, we discuss the C library functions that can be used to create and process disk files.

This chapter covers the following topics:

- 12.1 Streams and Buffers
 - 12.2 Files
 - 12.3 Accessing a File
 - 12.4 Character Input/Output
 - 12.5 String Input/Output
 - 12.6 Formatted File Input/Output
 - 12.7 Binary and Block I/O
 - 12.8 Random Access
 - 12.9 Other I/O Functions
-

12.1 Streams and Buffers

Programs need to perform input and output (I/O). Program input can come from input devices such as the keyboard and disk files. Program output can be sent to output devices such as the display, printer and disk files.

Stream I/O

In C, all input and output operations are carried out by means of *streams* that go to or come out from a program. A stream is a sequence of characters. It connects a program to a device (or file) and transfers the data in (input stream) and out (output stream) from the program. C programs deal with stream of characters instead of I/O devices (or files) for different kinds of input/output. The program just treats input/output as a continuous stream of characters, which is device independent. This has the advantage that the programmer does not need to write specific input/output functions for different devices such as keyboard, display, printer, etc.

Figure 12.1 illustrates the concept of stream I/O. In C, every stream connects to a file that refers to either a disk file or a physical device used for input or output. When the stream is connected to a display, data can flow from the program to the display through the stream. When we call functions such as `printf()` and `puts()` in the program, the stream carries data from the program to the display. Similarly, if the stream is connected to a keyboard, then when we call functions such as `scanf()` or `gets()`, the stream will carry the input data from the keyboard to the program.

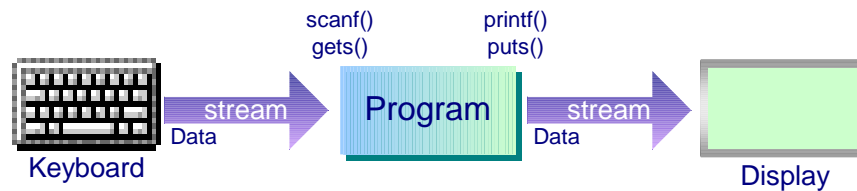


Figure 12.1 Stream I/O.

Standard Streams

Three standard streams are predefined in C. They are also known as standard input/output files. The standard streams are referred by three **FILE** pointers, namely **`stdin`** (standard input), **`stdout`** (standard output) and **`stderr`** (standard error). These streams are defined in the standard I/O header file *stdio.h*. The **FILE** pointer **`stdin`** is the standard input stream, which is normally connected to the keyboard. It is used by functions such as `scanf()`, `getchar()` and `gets()` that do not need to specify a **FILE** pointer, and the **`stdin`** is used as the default for input. The **FILE** pointer **`stdout`** is the standard output stream, which is normally connected to the display. It is used by functions such as `printf()`, `putchar()` and `puts()`. The **FILE** pointer **`stderr`** is the standard error stream, which is normally connected to the display. It is used by the operating system to display

error messages. Table 12.1 lists the three standard streams.

Table 12.1 The standard C streams.

S	F	D
tr	I	e
e	L	vi
a	E	c
m	P	e
	oi	
	n	
	te	
	r	
standard input	stdin	input - normally the keyboard
standard output	stdout	output - normally the display
standard error	stderr	error message - the display

Figure 12.2 shows the three standard streams, **stdin**, **stdout** and **stderr**, which are predefined in C. They are opened automatically by the system when a C program starts execution.



Figure 12.2 The three standard streams in C.

Stream Redirection

A C program normally uses the standard streams **stdin** and **stdout** as its sources for input and output. There are two ways for a C program to work with files. The first way is to use file I/O functions such as **fopen()**, **fclose()**, **fgets()**, **fputs()**, etc. This will be discussed later in this chapter. Another way is to use file redirection, which allows a program designed to use keyboard and display to redirect the input and output to and from files. This has the advantage that the program that uses functions such as **getchar()**, **scanf()**, **putchar()** and **printf()** can be run with file input and output without the need for modification.

File redirection is very useful for debugging interactive programs. An interactive program interacts with users for input data. Assume that a complex program needs many lines of input for data. The program must be tested thoroughly before it is considered fully operational. To test the program, the most convenient way is to enter all the test data into a file, and make the program read from that file as if they were entered by the user through the keyboard. It is also very useful to have the output recorded in a file for checking the correctness of the program. This can be achieved through file redirection.

Both Unix and DOS support file redirection. For example, assume we want to run the executable program called **prog** in a Unix system. The following command specifies input redirection:

```
$ prog < inputFile
```

The **\$** sign is the system prompt. The program **prog** accepts its input from the file **inputFile** and puts its output on the display. This is particularly useful for program testing when a lot of data need to be tested.

The following command specifies output redirection:

```
$ prog > outputFile
```

The program **prog** accepts its input from the users interactively. The output is sent to the **outputFile**. Figure 12.3 shows the output redirection.

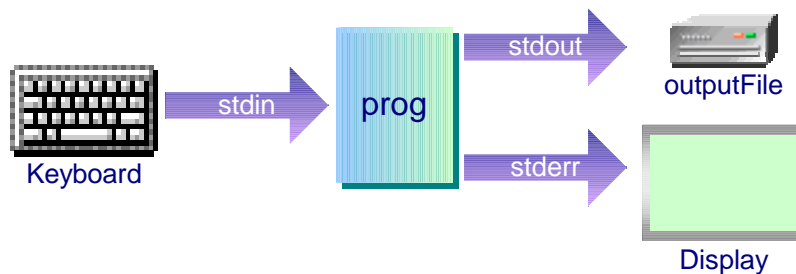


Figure 12.3 Output redirection: `prog > outputFile`.

The following command specifies the combined redirection:

```
$ prog < inputFile > outputFile
```

The program **prog** accepts its input from **inputFile** and sends its output to **outputFile**. This is fully non-interactive, as both input and output are directed to files. Figure 12.4 shows the combined redirection.

Buffering

Buffering is used by many C library functions for input and output. A buffer is a sequence of memory locations that is used to store data temporarily between the program and an I/O device. In computer, I/O operation is the most time consuming operation because it involves mechanical movement of the read/write head on the hard disk for each reading or writing. One can save a lot of precious computing time by doing less I/O by reading or writing block of characters each time instead of just one character at a time. By default, each program delays the real writing by

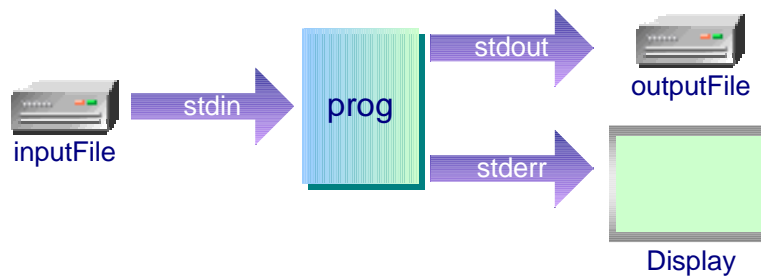


Figure 12.4 Combined redirection: `prog < inputFile > outputFile`.

sending the outputs to the buffer instead of to a file directly.

Figure 12.5 shows the buffered input/output. An input buffer is used to collect the input characters in sequence until the **<Enter>** key is pressed or a newline character is encountered when an input function such as `scanf()` or `gets()` is performed. Buffered input enables users to correct any typing mistakes before pressing the **<Enter>** key. Similarly, an output buffer is used to keep the output characters whenever output functions such as `printf()`, `putchar()` and `puts()` are performed. The data are transferred to the output device as a block whenever the output stream is flushed. This may occur when a newline character is encountered, or when the program switches from generating output to reading input.

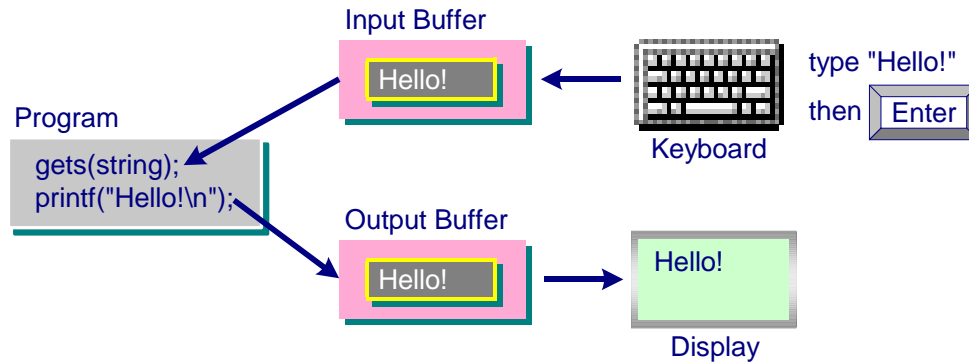


Figure 12.5 Buffered I/O.

In addition, the `fflush()` function (with the stream as its only argument) can also be used to flush the buffer. The argument for the `fflush()` is the **FILE** pointer or stream. The statements

```
printf("This is a test!");
fflush(stdout);
```

will flush the `stdout` and write the string "This is a test!" to the display immediately.

Some systems support unbuffered I/O functions which make input characters available to the program immediately. The unbuffered I/O functions execute faster than those I/O functions with buffered I/O. Unlike `stdout` and `stdin`, `stderr` is an unbuffered output stream. However, systems that support unbuffered I/O are closely tied to the operating systems and are less portable. For example, Borland C has the function called `getche()` which supports unbuffered I/O. It is impossible to standardize unbuffered I/O functions.

12.2 Files

C performs file input and output by means of streams. Disk file streams work in a similar way as the standard input/output streams. In C, a file stream can be viewed as a continuous sequence of bytes, and each of which can be accessed individually.

Binary and Text Modes

The two modes of access for disk files are text mode and binary mode. The text mode is used for text files in which data is stored and accessed as a sequence of

ASCII characters. While the binary mode is used for binary files in which data is stored and accessed in binary format.

When the text mode is used, translation on the end-of-line indicator occurs between C and the operating systems. For example, the MS-DOS system uses the carriage-return and linefeed combination (`\r\n`) to represent end-of-line, while C uses the newline (`\n`) to represent end-of-line. Therefore, when data is read from a disk file, each `\r\n` is translated to a `\n`. When data is written to a file, each `\n` is translated to `\r\n`. However, no translation is required when using text mode on Unix systems, as Unix also uses the newline (`\n`) to represent end-of-line. In binary mode, all data is read and written from and to the disk file using the same format. Thus, no translation is required between C and the operating systems when performing I/O operations.

End of File

To indicate the end of a file, one approach is to use an end-of-file marker that is a special character in the file. For example, Control-Z is used as the end-of-file marker in MS-DOS text file. Another approach is to store the size of the file in the operating system, so that the program knows the exact size of the file from the operating system. Unix systems adopt this approach. MS-DOS also uses this approach to store binary files.

In C, the value **EOF** (end-of-file) can be used to detect whether the end of a file is reached when using library functions such as `getchar()` and `scanf()`. **EOF** is defined in the header file *stdio.h* as follows:

```
#define EOF (-1)
```

In the statement

```
while ((ch = getchar()) != EOF) {  
    /* loop body */  
}
```

the return value from the `getchar()` function is compared with **EOF**. If they are different, i.e. the end of the file is not reached, then the statements inside the loop body will be executed.

12.3 Accessing a File

Apart from the standard streams **stdin**, **stdout** and **stderr** that are defined by default, we can define additional streams to read or write data from or to disk files. To access a file, we need to define a stream and open it for reading or writing. A

stream can be opened in text mode or binary mode. Program 12.1 shows the four steps involved in accessing disk files. The program opens two input files and one output file for processing.

Program 12.1 Accessing a disk file.

```
#include <stdio.h>

main(void)
{
    /* Step 1: Creating File Pointers */
    FILE    *fp1, *fp2, *fp3;
    int      i, j;

    /* Step 2: Opening files */
    fp1 = fopen("data1", "r");
    fp2 = fopen("data2", "r");
    fp3 = fopen("output", "w");

    /* Step 3: Performing I/O Operations */
    fscanf(fp1, "%d", &i);
    fscanf(fp2, "%d", &j);
    fprintf(fp3, "%d %d\n", i, j);

    /* Step 4: Closing Files */
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
    return 0;
}
```

Creating a File Stream

C programs may have several files open concurrently at any time. To use a particular file, we need to define a **FILE** pointer to the corresponding file. A new data type **FILE** is defined in the standard input/output header *stdio.h*. **FILE** is a structure data type which contains all the necessary information needed for using a file in C. We may treat the **FILE** pointer type **FILE *** as a stream. As has been discussed, **FILE** pointer is used to identify the file on which I/O operations can be performed. For example, we can create **FILE** pointers for the three files as

```
FILE *fp1, *fp2, *fp3;
```

where each of the **FILE** pointer variables **fp1**, **fp2** and **fp3** represents a file stream.

Opening a File Stream

We need to open a file using the **fopen()** function before we can perform any I/O operations on it. The function prototype of **fopen()** is

```
FILE *fopen(const char *filename, const char *mode);
or FILE *fopen(const char *, const char *);
```

fopen() returns a pointer to a file. It returns the **NULL** pointer if it cannot open the file. It takes in two input strings. The first string is the name of the file to be opened and the second string is the access mode. Table 12.1 lists the meaning of different access modes that can be specified. The access modes "**r**", "**w**" and "**a**" are respectively for reading, writing and appending data from or to the file. We need to be careful when an existing file is opened with "**w**" or "**w+**" mode, since the existing content of the file will be destroyed. For access modes "**r+**", "**w+**" and "**a+**", the file is opened for both reading and writing.

The following three statements

```
fp1 = fopen("data1", "r");
fp2 = fopen("data2", "r");
fp3 = fopen("output", "w");
```

Table 12.2 Access modes for a file.

Meaning	Access Mode					
	r	w	a	r+	w+	a+
File must exist before open	y	-	-	y	-	-
Old file deleted	-	y	-	-	y	-
File can be read	y	-	-	y	y	y
File can be written	-	y	y	y	y	y
File written only at the end	-	-	y	-	-	y

open the files **data1** and **data2** for reading and the file **output** for writing. The values that are returned by **fopen()** are assigned to the **FILE** pointer variables **fp1**, **fp2** and **fp3** respectively. In fact, the returned pointer such as **fp1** does not point to the actual file. It points to the **FILE** structure that contains all the information about the file maintained by the operating system.

The C library also supports both text and binary modes of files. To do this, we can append '**t**' to an access mode to specify the access is for text files, and '**b**' to specify the access for binary files. For example, the string "**rt**" is used for reading

a file in text mode. Similarly, the string `"rb"` is used for reading a file in binary mode. Other access modes for binary files are `"wb"`, `"ab"`, `"rb+"`, `"r+b"`, `"wb+"`, `"w+b"`, `"ab+"` and `"a+b"`. The string `"ab+"` is equivalent to `"a+b"`.

Performing Read/Write Operations

After the files have been opened and associated with the **FILE** pointer variables, we can perform read/write operations using these **FILE** pointers:

```
fscanf(fp2, "%d", &j);
fprintf(fp3, "%d %d\n", i, j);
```

where `fscanf()` and `fprintf()` are similar to `scanf()` and `printf()`. The `fscanf()` function takes inputs from the file pointed to by `fp2` and the `fprintf()` function writes output to the file pointed to by `fp3`. Table 12.3 lists some of the C functions that support I/O operations for the standard I/O streams and files.

Table 12.3 C library functions for I/O operations.

Functions to Access Standard Files	Functions to Access any Files
<code>printf, scanf,</code> <code>getchar, putchar,</code> <code>gets, puts</code>	<code>fprintf, fscanf,</code> <code>getc, fgetc, putc, fputc</code> <code>fgets, fputs</code> <code>fread, fwrite</code> <code>ftell, fseek</code> <code>ungetc, fflush</code>

Table 12.4 further classifies the I/O functions based on the different levels of I/O operations. The classification is mainly based on character, string and block levels of data access. The `fflush()` function has been discussed. The other file I/O functions will be discussed in detail later in this chapter.

Table 12.4 Classification of I/O library functions.

I/O Operations	C Library Functions
I/O on one character	<code>getc, fgetc, getchar, putc, fputc,</code> <code>putchar, ungetc</code>
I/O on more than one character	<code>scanf, fscanf, printf, fprintf</code>
I/O on one line of characters	<code>gets, fgets, puts, fputs</code>

I/O on one block of characters **fread, fwrite**

Closing a File Stream

After all the file I/O operations have been performed, the file should be closed. Although all files are automatically closed when a C program terminates, it is a good practice to close the files in the program. This is done by using the **fclose()** function, provided in the `<stdio>` library. The statement

```
fclose(fp2);
```

will close the file identified by **fp2**. The **fclose()** function returns 0 if successful, and **EOF** if not. The **fclose()** will fail if the disk is full or an I/O error has occurred.

Program 12.2 gives an example on accessing disk files using command line arguments. In the program, it opens 2 input files to read and one output file to write. The program uses command line arguments. The two input files are opened with "r" access mode, and the output file is opened with "w" access mode. The functions **fscanf()** and **fprintf()** work similarly to **scanf()** and **printf()**, except that a **FILE** pointer is required as its first argument. In addition, we also use **FILE** pointer of **stderr** to send out error message to the standard error. This program opens 2 input files, and one output file for I/O operations. There is a limit on how many files that can be opened at one time. This limit is usually 10 to 20, depending on systems. To run the executable program **prog**, we can type:

```
$ prog input1 input2 output
```

where **input1** and **input2** are the input filenames while **output** is the output filename.

Program 12.2 Accessing a disk file using command line arguments.

```
#include <stdio.h>

main(int argc, char *argv[])
{
    /* Step 1 */
    FILE    *fp1, *fp2, *fp3;
    int     i, j;

    if (argc != 4) {
```

```

    fprintf(stderr, "Usage: %s input1 input2 output\n",
        argv[0]);
    return(0);
}

/* Step 2 */
fp1 = fopen(argv[1], "r");
fp2 = fopen(argv[2], "r");
fp3 = fopen(argv[3], "w");

/* Step 3 */
fscanf(fp1, "%d", &i);
fscanf(fp2, "%d", &j);
fprintf(fp3, "%d %d\n", i, j);

/* step 4 */
fclose(fp1);
fclose(fp2);
fclose(fp3);
return 0;
}

```

12.4 Character Input/Output

The two functions `fgetc()` and `fputc()` support input/output at the character level. Table 12.5 lists the character input/output functions.

Table 12.5 Character input/output functions.

Function	Description	Examples (<code>FILE *fp; char ch;</code>)
<code>fgetc(fp)</code>	reads a character <code>ch</code> from the stream <code>fp</code> .	<code>ch = fgetc(fp);</code>
<code>getc(fp)</code>	reads a character <code>ch</code> from the stream <code>fp</code> .	<code>ch = getc(fp);</code>
<code>getchar()</code>	reads a character <code>ch</code> from <code>stdin</code> .	<code>ch = getchar();</code>
<code>fputc(ch, fp)</code>	writes a character <code>ch</code> to the stream <code>fp</code> .	<code>fputc(ch, fp);</code>
<code>putc(ch, fp)</code>	writes a character <code>ch</code> to the stream <code>fp</code> .	<code>putc(ch, fp);</code>

putchar(ch)	writes a character ch to	putchar(ch);
	stdout.	

The fgetc(), getc() and getchar() functions

The function **fgetc()** reads a character from a specified stream. The function prototype is

```
int fgetc(FILE *fp);
```

The function expects a **FILE** pointer of the file to read. It returns the next character in the specified file, or **EOF** if the end of the file is reached or an error has occurred. The **EOF** is defined in the header file *stdio.h*.

The function **getc()** is the same as **fgetc()** except that **getc()** is implemented as a macro. The function **getchar()** is also a macro (see Chapter 14) that takes no argument and reads a character from the standard input. It returns the next character, or **EOF** if the end of the file is reached or if there is an error.

The fputc(), putc() and putchar() functions

The function **fputc()** writes a character to a specified stream. It expects two arguments: a character to write and a pointer to the file to write. The function **putc()** is the same as **fputc()**. **putc()** is implemented as a macro. The function **putchar()** function is also a macro that expects a character to write to the standard output.

Program 12.3 shows a program, which reads the user input on filenames and opens the two specified files accordingly. The first file is opened for reading and the other is opened for writing. After the files are opened successfully, the characters are read in one by one from the first file, and then copied to the other file. At the same time, the characters read are also output to the **stdout**, i.e. the display. The reading stops when the **EOF** marker is encountered in the first file.

Program 12.3 Copying file data using character I/O functions.

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

```
main(void)
```

```
{
```

```
    char    filename1[80], filename2[80];
```

```
    FILE    *fp1, *fp2;
```

```
    char    ch;
```

```

printf("Enter the input file name: ");
gets(filename1);
printf("Enter the output file name: ");
gets(filename2);
fp1 = fopen(filename1, "r");
fp2 = fopen(filename2, "w");

if ((fp1 == NULL) || (fp2 == NULL)) {
    printf("Error in opening the files\n");
    exit(1);
}
while ((ch = fgetc(fp1)) != EOF) {
    fputc(ch, fp2);
    putchar(ch);
}
fclose(fp1);
fclose(fp2);
return 0;
}

```

12.5 String Input/Output

In some applications, it is more efficient to handle input/output in strings than characters. The function **fgets()** reads the whole string from a file, and the function **fputs()** writes a string to a file. In addition, the macro **gets()** reads a string from **stdin**, and the macro **puts()** writes a string to **stdout**. Table 12.6 lists some of the string input/output functions.

Table 12.6 String input/output functions.

Function (char *buf; int max; FILE *fp)	Description	Remarks on the newline and null character
fgets(buf, max, fp)	reads string from a file and stores it into buf from stream fp . At most max-1 characters are read.	retains newline character and adds null character.
gets(buf)	reads characters from stdin and places them into buf .	replaces newline character with null character.

fputs(buf, fp)	writes the string pointed to by buf to the stream fp .	does not add null character and newline character.
puts(buf)	writes the string pointed to by buf to stdout .	adds newline character, does not add null character.

The fgets() and gets() functions

The **fgets()** function expects three arguments. The first argument is the address of the character array where input should be stored. The second argument is the maximum size of the string to store. The third argument is the **FILE** pointer, which identifies the file to be read. If **max** is the maximum size of the string, the **fgets()** function reads characters from the file until all characters up to and including the first newline character have been read, or **max-1** characters have been read, or end-of-file is reached. If **fgets()** reads a newline character, it is retained and stored at the end of the string. In addition, the terminating null character (`'\0'`) is also added to the end of the string. The **fgets()** function never stores more than the specified maximum size including the newline and the terminating null character. As such, it is guaranteed that **fgets()** will not store the string exceeding the size of the array. This makes **fgets()** a better function to use than **gets()**. **fgets()** returns the value **NULL** when the **EOF** marker is encountered. Otherwise, the address of the character array used to store the string is returned.

The function **gets()** is similar to **fgets()**. However, it differs from **fgets()** in the following ways. **gets()** reads a string from **stdin**, no **FILE** pointer is needed. It only expects one argument, the address of the array in which the string is to be stored. Moreover, the maximum size of the string is not specified. **gets()** reads until it encounters a newline or end of file. But the newline is discarded rather than stored in the array. **gets()** returns **NULL** if it fails, otherwise it adds the null character (`'\0'`) and returns the address of the array.

The fputs() and puts() functions

The function **fputs()** writes a string of characters to a file. It expects two arguments. The first argument is the address of a string, and the second argument is a **FILE** pointer. **fputs()** does not add a newline character to the end of the string. The function **puts()** is similar to **fputs()** except that it expects only one argument on the address of a null-terminated array. **puts()** writes to the **stdout**. It also differs from **fputs()** in that it adds a newline character to the end of the string. Neither **fputs()** nor **puts()** copies the null terminating character (`'\0'`)

to the file. Both return **EOF** if they fail, otherwise the last character written is returned.

Similar to Program 12.3, Program 12.4 shows how to perform file I/O operations on strings using string I/O functions **fgets()** and **fputs()**. The **fgets()** function reads the number of characters specified by **MAX_SIZE-1** and stores them in **line[]**. The **fputs()** function then writes the string stored in **line[]** to the output file. At the same time, the same content is written to the **stdout**, i.e. the display.

Program 12.4 Copying file data using string I/O functions.

```
#include <stdio.h>
#define MAX_SIZE 80

main(void)
{
    char    filename1[80], filename2[80];
    FILE    *fp1, *fp2;
    char    line[MAX_SIZE];

    printf("Enter the input file name: ");
    gets(filename1);
    printf("Enter the output file name: ");
    gets(filename2);
    fp1 = fopen(filename1, "r");
    fp2 = fopen(filename2, "w");

    if ((fp1 == NULL) || (fp2 == NULL)){
        printf("Error in opening the files\n");
        exit(1);
    }
    while (fgets(line, MAX_SIZE, fp1) != NULL) {
        fputs(line, fp2);
        puts(line);
    }
    fclose(fp1);
    fclose(fp2);
    return 0;
}
```

12.6 Formatted File Input/Output

The functions `fprintf()` and `fscanf()` perform formatted input and output. They are similar to `printf()` and `scanf()` except that they require an additional first argument on **FILE** pointer to identify the file on which the I/O operation is to be performed. The format for `fprintf()` is

```
fprintf(fp, control-string, argument-list);
```

where `fp` is a **FILE** pointer to the file to be read. The other arguments of `fprintf()` are the same as those in `printf()`.

Similarly, the format for `fscanf()` is

```
fscanf(fp, control-string, argument-list);
```

where `fp` is a **FILE** pointer to the file to be written. The other arguments of `fscanf()` are the same as those in `scanf()`.

The `fprintf()` and `fscanf()` functions are operated in text mode, which will read data as strings of characters. This is not as efficient as in binary mode, as it requires more storage to store numeric data.

Program 12.5 shows a program using formatted file I/O functions `fprintf()` and `fscanf()`. The program first generates ten lines of data, and each line consists of an integer value and a floating point value. The data is then written to a file called "input". The program then opens the same file for reading. The ten lines of data are read using the `fscanf()` function. The data is then written to `stdout` using the `printf()` statement.

Program 12.5 Copying file data using formatted file I/O functions.

```
#include <stdio.h>

main(void)
{
    FILE    *fp1, *fp2;
    int     i,j,k;
    float   num1=1.0, num2;

    if ((fp1 = fopen("input", "w")) == NULL) {
        fprintf(stderr, "Error in opening the files\n");
        exit(1);
    }
    for (i=0; i<10; i++){
        num1 = num1*10.0;
```

```

    fprintf(fp1, "%d %f\n", i, num1);
}
fclose(fp1);

if ((fp2 = fopen("input", "r")) == NULL) {
    fprintf(stderr, "Error in opening the files\n");
    exit(1);
}
printf("The values are:\n");
for (j=0; j<10; j++){
    fscanf(fp2, "%d %f", &k, &num2);
    printf("%d %f\n", k, num2);
}
fclose(fp2);
return 0;
}

```

12.7 Binary and Block I/O

The functions `fread()` and `fwrite()` read and write binary data in blocks. Binary I/O is faster and more efficient than writing individual characters or strings. The data written may be of basic data types such as integers, floating points, arrays or structures. For example, numeric data may be stored in one of the two modes in a file: text mode or binary mode. To write an integer number in text mode, the statement

```
fprintf(fp, "%d", number);
```

can be used. The binary mode can also be used to write the integer number. The statement

```
fwrite(&number, sizeof(int), 1, fp);
```

can be used for writing data in binary mode.

Figure 12.6 illustrates the storage requirements for the text mode and binary mode. For example, an integer number 13579 can be stored as a sequence of 5 characters in text mode, but it can be represented in binary format for just 2 bytes (assuming that 2 bytes are used for `int`) in binary mode. As integer number is represented as a string of characters in text mode, the size of storage required depends on the value of the number. For binary mode, it depends on the systems, as different systems have different storage requirements for their data types. Thus, to store the data, the text mode needs 5 bytes, while the binary mode only needs 2 bytes. Therefore, the use of binary mode helps to increase the disk storage

efficiency. If the file is stored in binary mode, it must be accessed in binary mode for I/O operations.

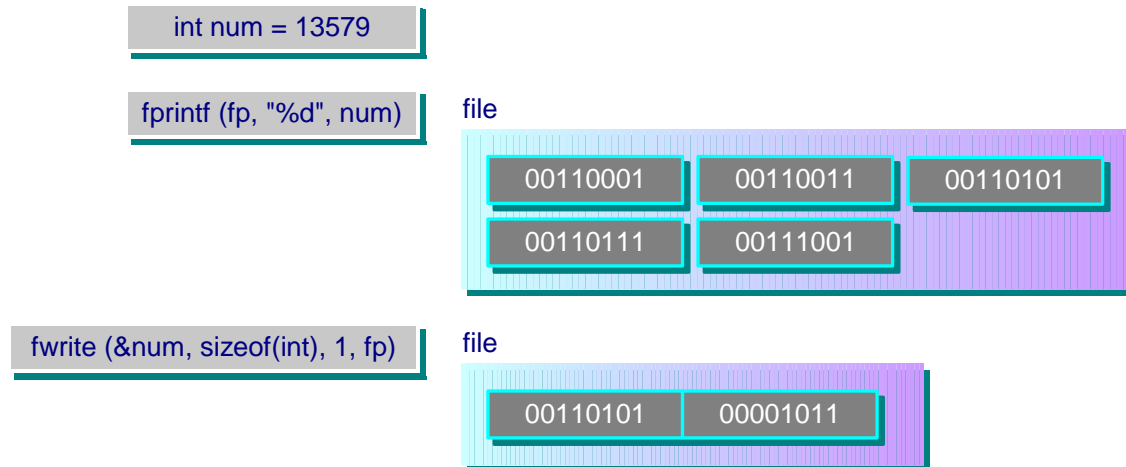


Figure 12.6 Storage requirements for text and binary modes for an integer.

The fread() function

The function `fread()` expects four arguments:

```
fread(ptr, size, num, fp);
```

`fread()` reads into the item pointed to by `ptr`, up to `num` elements whose size is specified by `size`, from the file pointed to by the `FILE` pointer `fp`. `fread()` returns the number of items successfully read, which may be less than `num` when end-of-file is encountered or there is an error. The statements

```
double array[10];
fread(array, sizeof(double), 10, fp);
```

read in binary form the 10 elements of the `array`, each with size `double` to the file pointed to by `fp`.

The fwrite() function

The function `fwrite()` also expects four arguments:

```
fwrite(ptr, size, num, fp);
```

fwrite() writes from the address pointed to by **ptr**, up to **num** elements whose size is specified by **size**, to the file pointed to by **fp**. **fwrite()** returns the number of items successfully written, which will be less than **num** only if a write error is encountered. The statements

```
char array[80];
fwrite(array, 1, 80, fp);
```

write in binary form the 10 elements of the **array**, each with size **char** to the file pointed to by **fp**.

Program 12.6 shows how to use binary and block I/O functions to perform the **fwrite()** operation on complex data structure. In the program, the structure **component** is first defined using **typedef**. The structure consists of 4 elements including the name of the component, the serial code of the component, the amount or quantity of the component, and the cost of the component. This information is stored in a file for inventory purposes. The **FILE** pointer is then declared. The name of the file to be used to store the inventory information is obtained from user input. It is then opened in binary mode using access mode "**wb**". The user also enters the total number of components to be stored. Once this is done, the program starts reading the component information one by one. The **fwrite()** function

```
fwrite(&comp_data, sizeof(comp_data), 1, fp);
```

enables the whole block of **comp_data** to be written in binary format to the file.

Program 12.6 Writing data into a file using block I/O functions.

```
#include <stdio.h>

typedef struct {
    char   name[20];
    int    serial_code;
    int    amount;
    int    cost;
} component;

main(void)
{
    component comp_data;
    FILE      *fp;
    char      filename[80];
    int       numofcomp;
```

```
int      i;

printf("Enter the file name: ");
gets(filename);
if ((fp = fopen(filename, "wb")) == NULL) {
    printf("can't open file \n");
    exit(1);
}
printf("Enter the number of components: ");
scanf("%d", &numofcomp);
for (i=0; i<numofcomp; i++) {
    printf("Name of the component: ");
    fflush(stdin);
    gets(comp_data.name);
    printf("Serial code of the component: ");
    scanf("%d", &comp_data.serial_code);
    printf("Amount of the component: ");
    scanf("%d", &comp_data.amount);
    printf("Cost of the component: ");
    scanf("%d", &comp_data.cost);
    fwrite(&comp_data, sizeof(comp_data), 1, fp);
    printf("\n");
}
fclose(fp);
return 0;
}
```

Program input and output

```
Enter the file name: comp.dat
Enter the number of components: 2
Name of the component: comp1
Serial code of the component: 1234
Amount of the component: 45
Cost of the component: 23

Name of the component: comp2
Serial code of the component: 2345
Amount of the component: 167
Cost of the component: 28
```

Program 12.7 shows how to use binary and block I/O functions to perform the reading operation on the complex data structure **component**. The input file will be read from the user and is opened in binary mode using access mode "**rb**". The

program then reads in the structure `comp_data` using the `fread()` function. The `fread()` function will return the number of items successfully read. So for each `fread()` operation, the number of items is 1 as we set `num=1` in the `fwrite()` function. When the number of items read is not 1, it means that the file cannot be read anymore, then the `while` loop will exit.

Program 12.7 Reading data from a file using block I/O functions.

```
#include <stdio.h>

typedef struct {
    char   name[20];
    int    serial_code;
    int    amount;
    int    cost;
} component;

main(void)
{
    component comp_data;
    FILE      *fp;
    char      filename[80];

    printf("Enter the file name: ");
    gets(filename);
    if ((fp = fopen(filename, "rb")) == NULL) {
        printf("can't open file \n");
        exit(1);
    }
    while (fread(&comp_data, sizeof(component), 1, fp) == 1){
        printf("Name of the component: %s\n", comp_data.name);
        printf("Serial code: %d\n", comp_data.serial_code);
        printf("Amount: %d\n", comp_data.amount);
        printf("Cost: %d\n", comp_data.cost);
        printf("\n");
    }
    fclose(fp);
    return 0;
}
```

Program input and output

```
Enter the file name: comp.dat
Name of the component: comp1
Serial code: 1234
```

```
Amount: 45
Cost: 23

Name of the component: comp2
Serial code: 2345
Amount: 167
Cost: 28
```

12.8 Random Access

All the previous I/O functions perform reading and writing on files sequentially. Data are written to the file in order and data are read from the file in the order they are stored. The functions `fseek()`, `ftell()` and `rewind()` help performing I/O in a non-sequential manner. It can read the 10th datum, then go back and read the 2nd datum. This is called *random access*.

Figure 12.7 illustrates the concept in random file access. To support random access, the system keeps a *file position marker* for each open file. The file position marker indicates the location in the file at which data will be read or written. When we access the file for read operation, the file position marker starts at position 0. When we access the file for write operation, the file position marker is located at position 0. When we access the file for the append operation, the file position marker is first located at the end of the file before new data is added to the file. Therefore, the file position marker is initialized at the beginning of the file if the file is opened with access mode "r", "w", "r+" or "w+". The file position marker is located at the end of the file if the file is opened with access mode "a" or "a+".

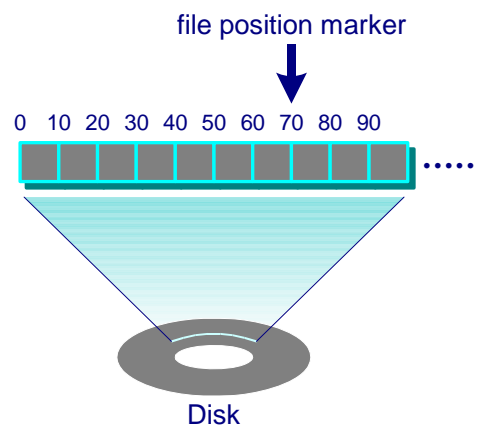


Figure 12.7 Random file access with a file position marker.

The `fseek()` Function

The `fseek()` function moves the file position marker to a specified location. The function prototype of `fseek()` is

```
int fseek(FILE *fp, long offset, int mode);
```

The **offset** tells how far the file position marker moves from the starting point (depending on the mode). It can be positive (indicates to move forward), negative (indicates to move backward) or 0. The **mode** identifies the starting point. Table 12.7 shows the constants defined in *stdio.h* that can be assigned to **mode**. `fseek()` returns 0 if it is successful, and -1 if there is an error. If the file position marker is reset, then the subsequent read/write operations will start from the new location.

Table 12.7 Constants for defining mode.

Mode	Value	Meaning
SEEK_SET	0	Measured from the beginning of the file.
SEEK_CUR	1	Measured from the current position.
SEEK_END	2	Measured from the end of the file.

For example, the statement

```
fseek(fp, 0, SEEK_SET);
```

sets the file position marker at the beginning of the file. The statement

```
fseek(fp, 100L, SEEK_CUR);
```

sets the file position marker 100 bytes (toward the end of the file) from the current file marker position. The statement

```
fseek(fp, -100L, SEEK_END);
```

sets the file position marker 100 bytes (toward the beginning of the file) from the end of the file.

The `ftell()` Function

The `ftell()` function returns the location of the file position marker in bytes measured from the beginning of the file. The function prototype of `ftell()` is

```
long ftell(FILE *fp);
```

which expects one argument on **FILE** pointer. The function returns -1 if there is an error. **ftell()** applies to files opened in binary mode. In the case of text files, **ftell()** is not that useful. It is because text files may contain characters entered by the systems rather than by the programs. For example, in MS-DOS text files, the **\r\n** is used to represent end-of-line, while C uses **\n** to represent end-of-line. Therefore, we cannot use **ftell()** to determine exactly the number of non-system characters between the beginning of the file and the file position marker.

The **rewind()** Function

The **rewind()** function resets the file position marker to the beginning of the file. The function prototype of **rewind()** is

```
void rewind(FILE *fp);
```

This is equivalent to

```
fseek(fp, 0L, SEEK_SET);
```

Program 12.8 and Program 12.9 give two examples on processing files using random file access. Although the files opened in the two programs are in binary mode, random file access can also be used in files opened with text mode. Program 12.8 shows a sample program for random file access. The file is opened in binary mode with access mode **"rb"**. The program reads in the record number of the component from the user. The position of the record **rec_position** is then computed. It then moves the file position marker to the record position using the **fseek()** function from the beginning of the file.

After the statement

```
fseek(fp, rec_position, SEEK_SET);
```

is executed, the record position is located. The structure data **comp_data** is then read from the file using the **fread()** function

```
fread(&comp_data, sizeof(comp_data), 1, fp);
```

The individual members of the structure are then printed onto the screen.

Program 12.8 Reading data from a file using random file access.

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

```
typedef struct {
```

```
    char  name[20];
    int    serial_code;
    int    amount;
    int    cost;
} component;

main(void)
{
    component  comp_data;
    FILE       *fp;
    char       filename[80];
    int        rec_number;
    long       rec_position;

    printf("Enter the file name: ");
    gets(filename);
    if ((fp = fopen(filename, "rb")) == NULL) {
        printf("can't open file \n");
        exit(1);
    }
    printf("Enter the record number of component: ");
    scanf("%d", &rec_number);
    rec_position = (rec_number-1) * sizeof(comp_data);
    fseek(fp, rec_position, SEEK_SET);
    if ((fread(&comp_data, sizeof(comp_data), 1, fp)) == 1){
        printf("Name of the component: %s\n", comp_data.name);
        printf("Serial code: %d\n", comp_data.serial_code);
        printf("Amount: %d\n", comp_data.amount);
        printf("Cost: %d\n", comp_data.cost);
    }
    fclose(fp);
    return 0;
}
```

Program input and output

```
Enter the file name: comp.dat
Name record number of the component: 2
Name of the component: comp2
Serial code: 2345
Amount: 167
Cost: 28
```

Program 12.9 shows another example that copies file data from the input file to the output file in a reverse order using random file access I/O functions **fseek()**

and `ftell()`. The file is opened in binary mode with access mode `"rb"`. The statement

```
fseek(fp, 0L, SEEK_END);
```

moves the current file marker position to the end of the file. The statement

```
total = ftell(fp);
```

assigns the number of bytes from the beginning to the end of the file to the variable `total`. Then the `for` loop will copy and print the characters one by one while the file position marker moves backward one character at a time using the statement

```
fseek(fp, -i, SEEK_END);
```

The loop stops when the first character from the beginning of the file has been printed.

Program 12.9 Copying file data in a reverse order using random file access.

```
#include <stdio.h>

main(void)
{
    char    filename1[80], filename2[80];
    char    ch;
    FILE    *fp1, *fp2;
    long    i, total;

    printf("Enter the input file name: ");
    gets(filename1);
    printf("Enter the output file name: ");
    gets(filename2);
    fp1 = fopen(filename1, "rb");
    fp2 = fopen(filename2, "wb");

    if ((fp1 == NULL) || (fp2 == NULL))
        printf("can't open files \n");
    else {
        fseek(fp1, 0L, SEEK_END); /* move file marker position
                                   to EOF */
        total = ftell(fp1);
        for (i=1L; i<=total; i++) {
```

```
        fseek(fp1, -i, SEEK_END);
        ch = fgetc(fp1);
        if ((ch != EOF)) {
            fputc(ch, fp2);
            putchar(ch);
        }
    }
}
fclose(fp1);
fclose(fp2);
return 0;
}
```

12.9 Other I/O Functions

Apart from the above I/O library functions, we discuss two other I/O functions: `ungetc()` and `feof()`.

The `ungetc()` Function

The function `ungetc()` reverses the operation done by `getchar()`, `getc()` and `fgetc()`. The function prototype is

```
int ungetc(int c, FILE *fp);
```

For example, after a read operation (e.g. `getchar()`) to read a character from the input buffer, the `ungetc()` function can place the read character specified by `c` back to the input buffer.

The `feof()` Function

The `feof()` function returns a nonzero value if the end-of-file indicator is detected, and returns zero otherwise. The function prototype is

```
int feof(FILE *fp);
```

This is mainly used to check whether the end-of-file has been reached. The statement

```
while ( !feof(fp) ) {
    /* loop body */
}
```

can be used to test the end-of-file condition. The loop body in the **while** loop will be executed if the end-of-file is not reached.

12.10 Exercises

1. Explain the following terms: stream I/O, standard streams and stream redirection.
2. Let "sc103.dat" be a file of records containing information on SC103 students. Each record contains a student ID, his/her scores for lab 1 to 10, and his/her final examination score. A sample of the sc103.dat is given as follows:

2002A01	60 80 70 50 40 67 78 89 86 34 78
2002A02	50 90 80 70 80 88 56 78 84 89 83
.....	

Assume that the following structure is used to define a student record:

```
typedef struct {
    char  id[10];
    int   lab[5];
    int   exam;
} studentRec;
```

Write a C program that reads each record from the file, computes the final score, and prints the student ID together with the final score on the display. The final score is computed based on the following formula:

$$\text{Final score} = (\text{total lab scores})/10*0.2 + \text{final examination score}*0.8$$

3. Write a C program `printNlines` that takes two command line arguments. The first is a non-negative integer `n`, and the second is a filename. The format for the command is: `printNlines n filename`. The program prints only the first `n` lines in the file. Assume that all the lines can be found in the file, and each line is no longer than 100 characters.
4. Write a C program that reads from a file a line of alphabetic text, converts it into uppercase letters and writes it back to the same file. The format for the command is: `convertupper filename`. Assuming that the line contains at most 100 characters.



Storage Classes

Every variable and function has a storage class. The storage class determines the visibility of a variable or function, and the location in the program where it can be made reference to. Four keywords can be used to define storage classes: **auto**, **extern**, **static** and **register**. This chapter discusses the various storage classes. We also introduce type qualifiers, which inform the compiler on how to access a variable.

This chapter covers the following topics:

- 13.1 Storage Classes
 - 13.2 Storage Class **auto**
 - 13.3 Storage Class **extern**
 - 13.4 Storage Class **static**
 - 13.5 Storage Class **register**
 - 13.6 Local and Global Variables
 - 13.7 Nested Blocks with the Same Variable Name
 - 13.8 Storage Classes for Multiple Source Files
 - 13.9 Storage Classes for Functions
 - 13.10 ANSI C Type Qualifiers
-

13.1 Storage Classes

The storage class of a variable is a set of properties about the variable. It is determined by where it is defined and which keyword (i.e. **auto**, **extern**, **static** or **register**) it is used with. The storage class of a variable determines the scope, linkage and storage duration of the variable. The *scope* determines the

region of the code that can use the variable. In other words, the variable is visible in that section of code. The *linkage* determines how a variable can be used in a multiple-source-file program. It identifies whether the variable can only be used in the current source file or it can be used in other source files with proper declarations. The *storage duration* determines how long a variable can exist in memory.

Scope of Variables

A C variable has one of the following three scopes: file scope, block scope or function prototype scope. A variable has the *file scope* if the variable is visible until the end of the file containing the definition. A variable has the *block scope* if the variable is visible until the end of the block containing the definition. A variable has the *function prototype scope* if the variable is visible to the end of the prototype declaration.

Program 13.1 illustrates the scopes of variables. **global_var** has the file scope which is visible to the program until the end of the file. **var1** and **var2** have function prototype scope, which are only visible until the end of the function prototype declaration. **var3** and **var4** have block scope, which are visible until the end of the function block.

Program 13.1 Scopes of variables.

```
int global_var;          /* global_var has the file scope */
int prototype(int var1, int var2);
    /* var1 and var2 have function prototype scope */
main(void)
{
    ....
}
double function(double var3)    /* var3 has block scope */
{
    double var4=0.0;           /* var4 has block scope */
    ....
    return var4;
}
```

Linkage of Variables

A C variable has one of the following three linkages: external linkage, internal linkage and no linkage. A variable has *external linkage* if the variable can be used anywhere in a multiple-source-file program. A variable has *internal linkage* if the variable can be used anywhere in a file. If a variable can be used only within a block, the variable has *no linkage*. Variables with block scope or function

prototype scope have no linkage.

Storage Duration of Variables

The storage duration of a C variable can be static or automatic. A variable has *static* storage duration if the variable exists throughout the program execution. A variable has *automatic* storage duration if the variable only exists within a block where the variable is defined.

Program 13.2 illustrates the storage duration of variables. **global_var** has static storage duration, which exists throughout the program execution. **var1** and **var2** have automatic storage duration, which exist only during the execution of the function block **function1()**. The variables are created and memory locations are allocated when the function is called. When the function finishes execution, the memory locations are no longer available for use by the variables.

Program 13.2 Storage duration of variables.

```
#include <stdio.h>
int global_var = 5;
    /* global_var has static storage duration */
void function1(int var1);
main(void)
{
    function1(global_var);
    return 0;
}
void function1(int var1)
    /* var1 has automatic storage duration */
{
    int var2;          /* var2 has automatic storage duration */
    for (var2=0; var2<var1; var2++) {
        puts("Welcome to Introduction to C Programming");
    }
}
```

Program output

```
Welcome to Introduction to C Programming
Welcome to Introduction to C Programming
Welcome to Introduction to C Programming
Welcome to Introduction to C Programming
Welcome to Introduction to C Programming
```

13.2 Storage Class **auto**

Automatic (or **auto**) variables are declared with storage class keyword **auto** inside the body of a function or block. A block contains a complex statement that is enclosed by braces `{}`. We can define an automatic variable by using the storage class keyword **auto** as

```
auto int i, j, k;
```

or we can omit the keyword as

```
int i, j, k;
```

An automatic variable has automatic storage. The lifetime of an automatic variable is only within the function or block where it is defined. It has no meaning outside the function or block. It is also called the *local variable* of the corresponding function or block. Automatic variables are destroyed after the execution of the corresponding function or block where they are defined. It is because they are no longer needed and the memory occupied by the variables can be reused by other functions. Therefore, the value stored in an automatic variable is lost after exiting from the function or block.

An automatic variable has the block scope. Only the function in which the variable is defined can access that variable by name. Further, an automatic variable has no linkage. The variable cannot be declared twice in the same block.

Program 13.3 shows a function that defines automatic variables. Automatic variables can be defined as the function argument, e.g. **k** in the function, or they can be defined within a block, e.g. **i** and **j**.

Program 13.3 Automatic variables.

```
int function(int k)           /* k - automatic variable */  
{  
    int i;                   /* i - automatic variable*/  
  
    for (i=0; i<10; i++) {  
        int j;               /* j - automatic variable*/  
        for (j=0; j<4; j++)  
            if (k%j == 0) k++;  
    }  
    return k;  
}
```

13.3 Storage Class **extern**

An external (or **extern**) variable is defined outside a function's body. However, it can also be declared inside a function that uses it by using the **extern** keyword. External variables have static storage duration and external linkage. The storage is allocated to the variable when it is declared and it lasts until the end of program execution. If the variable is defined in another file, declaring the variable with the keyword **extern** is mandatory. External variables are initialized once when the storage is allocated. If the external variable is not explicitly initialized in the program, it will be automatically initialized to zero by the compiler.

Program 13.4 illustrates external variables. There are two declarations for **Earray**, the two declarations refer to the same variable **Earray**. The second declaration

```
extern double Earray[];
```

for the variable is optional. However, the declaration for the external variable **Eint** is mandatory in the first declaration

```
extern int Eint;
```

as the second **Eint** declaration is only declared after the **main()** function. The **main()** function cannot refer to the variable unless it is declared inside the **main()** as external variable. The variable **Echar** has been declared in another file. Thus the use of the keyword **extern** is mandatory.

External variables allow us to break down a large program into smaller files and compile each file separately. This helps to reduce program development time, as only those files that have been changed are required to be compiled again.

Program 13.4 External variables.

```
double Earray[10]; /* externally defined array */
extern char Echar; /* mandatory declaration */
                      /* Echar is defined in another file */
int next(void);

main(void)
{
    extern int Eint; /* mandatory declaration for main to use
                      Eint*/
    extern double Earray[]; /* optional declaration */
    Eint +=10;
    ....
}
```



```

}
int Eint;          /* externally defined variable */
void function(void)
{
    Eint++;        /* subsequent functions can use Eint */
    ....
}

```

13.4 Storage Class static

A *static* variable may be defined inside or outside a function's body. This section only considers static variables defined inside a function. Section 13.7 will discuss the static variables defined outside a function. The duration of a static variable is fixed. Static variables are created at the start of the program and are destroyed only at the end of the program execution. We can define static variables *inside* a function's body by changing an automatic variable using the keyword **static** as follows:

```
static int i;
```

If a static variable is defined and initialized, it is then initialized once when the storage is allocated. If a static variable is defined, but not initialized, it will be initialized to zero by the compiler. The initialization is done when the storage is allocated. If the static variable is defined inside a function's body, then the variable is only visible by the block containing the variable. Static variables are very useful when we need to write functions that retain values between functions. We may use global variables to achieve the same purpose. However, static variables are preferable as they are local variables to the functions, and the shortcomings of global variables can be avoided.

Program 13.5 illustrates the use of static variables. The static variable **static_var** is declared and initialized only once when storage is allocated at the start of the program. The value of the static variable is retained for different calls to **function()**. The value stored in the static variable will remain until the end of the program execution. This is different from automatic variable as it is created and initialized every time **function()** is called. However, since the static variable **static_var** is declared inside **function()**, it is only visible inside **function()**.

Program 13.5 Static variables.

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

```
void function();
main(void)
{
    int i;
    for (i=0; i<3; i++) function();
    return 0;
}
void function()
{
    static int static_var = 0;    /* static variable */
    int auto_var = 0;            /* automatic variable */

    ++static_var;
    ++auto_var;
    printf("The value for the static variable is %d\n",
        static_var);
    printf("The value for the automatic variable is %d\n",
        auto_var);
}
```

Program output

```
The value for the static variable is 1
The value for the automatic variable is 1
The value for the static variable is 2
The value for the automatic variable is 1
The value for the static variable is 3
The value for the automatic variable is 1
```

13.5 Storage Class register

An automatic variable can be defined using the keyword **register** as shown in Program 13.6. It informs the compiler that the variables will be made reference to on numerous occasions and, where possible, to use the CPU registers. Instructions using register variables execute faster than instructions that use no register variables. However, only a limited number of registers are available. The use of register variables is applicable to automatic variables and function arguments only and is restricted to certain data types (which is machine dependent) such as **int** and **char**. Pointers are not allowed to take the address of register variables, i.e. **&** operator will not work with register variables.

Program 13.6 Register variables.

```
int function(register int k)    /* register variable */
```

```
{
    register int i;                /* register variable */

    for (i=0; i<10; i++) {
        register int j;          /* register variable */

        for (j=0; j<4; j++)
            if (k%j==0) k++;
    }
    return k;
}
```

13.6 Local and Global Variables

Local variables are variables defined inside a function. They have function or block scope. They can be accessed only within the function. They cannot be accessed by other functions. Local variables are created when the function is invoked, and destroyed after the complete execution of the function.

Global variables are variables defined outside the functions. They have file (or program) scope. Thus, global variables are visible to all the functions that are defined following its declaration. Program 13.7 illustrates the local and global variables.

The advantages of global variables in programs are that global variables are the simplest way of communication between functions and they are efficient. The disadvantages of programs using global variables are that they are less readable and more difficult to debug and modify as any functions in the program can change the value of the global variable. Therefore, it is considered good programming practice to use local variables, and use parameter passing between functions for communication between functions. In this way, the value of each variable in the function is protected.

Program 13.7 Local and global variables.

```
#include <stdio.h>
void function();
int global_var;
main(void)
{
    global_var = 10;
    printf("The global variable is %d\n", global_var);
    function();
    return 0;
}
```

```

void function()
{
    int local_var = 5;
    global_var += local_var;
    printf("The local variable is %d\n", local_var);
    printf("The global variable in the function is %d\n",
        global_var);
}

```

Program output

```

The global variable is 10
The local variable is 5
The global variable in the function is 15

```

Table 13.1 summarizes the storage classes of different variables. Global variables are declared ahead of all the functions. Local variables are declared within the functions. Global variables can take the storage keyword **extern** or without the keyword. Local variables can take storage keywords **auto**, **register** and **static**, or without the storage keyword. The storage duration for global variables is static that exists until the program ends. For local variables, it may be automatic for local variables with keywords **auto** or **register**, or static for variables with the keyword **static**. Global variables can have external linkage if it is **extern** variables, or internal linkage for global variables defined without any storage keyword.

Table 13.1 Classes of variables.

Variables	Keyword	Scope	Linkage	Storage Duration
automatic	auto	block	no	automatic
external	extern	file	external	static
static	static	file/block	internal	static
register	register	block	no	automatic
local	none/auto/ static/ register	block	no	automatic/static
global	none/extern	file	internal/external	static

If the global variables and local variables have the same name, then the variables within the function that defines the local variables refer to the local

variables, not the global variables. This is illustrated in Program 13.8.

Program 13.8 Local and global variables with the same name.

```
#include <stdio.h>
void function();
int x;
main(void)
{
    x = 10;
    printf("The global variable x is %d\n", x);
    function();
    printf("The final global variable x is %d\n", x);
    return 0;
}
void function()
{
    int x;
    x = 20;      /* this refers to the local variable x */
    printf("The local variable x is %d\n", x);
}
```

Program output

```
The global variable x is 10
The local variable x is 20
The final global variable x is 10
```

13.7 Nested Blocks with the Same Variable Name

Variables are allowed to have the same name as long as they are defined under different scope or block. A block is a group of statements enclosed by braces. A block may be nested within other blocks. Program 13.9 illustrates the scope of variables that use the same variable name. When the variables with the same name are declared in different nested blocks, the variables with smaller scope will temporary hide the other variables within that block.

In Program 13.9, the first `printf()` statement prints the value 20 for `i`. The second `printf()` statement prints the value 200 for `i` as it refers to the definition in block 2. The third `printf()` statement prints the value 400 for `i` as it refers to the definition in block 3. The last `printf()` statement prints the value of 30 as `i` refers back to the variable defined in block 1.

If the declaration of the variable `i` in line 12 in Program 13.9 is removed, the third `printf()` statement will print

Variable `i` in block 3 = 500

This is because the variable `i` in block 3 now refers to the `i` defined in block 2.

Program 13.9 Nested blocks with the same variable name.

```
#include <stdio.h>
main(void)
{
    /* begin block 1 */
    int i = 10;
    i += 10;
    printf("Variable i in block 1 = %d\n", i);
    {
        /* begin block 2 */
        int i = 100;
        i += 100;
        printf("Variable i in block 2 = %d\n", i);
        {
            /* begin block 3 */
            int i = 300;
            i += 300;
            printf("Variable i in block 3 = %d\n", i);
        }
        /* end block 3 */
    }
    /* end block 2 */
    i += 10;
    printf("Variable i in block 1 = %d\n", i);
    return 0;
}
/* end block 1 */
```

Program output

```
Variable i in block 1 = 20
Variable i in block 2 = 200
Variable i in block 3 = 600
Variable i in block 1 = 30
```

13.8 Storage Classes for Multiple Source Files

Program 13.10 illustrates storage classes for a multiple-source-file program. The program consists of two files. We can declare a variable with keywords such as **extern** and **static** outside functions. In the following declaration in *file1.c*:

```
int index1;
```

the variable **index1** is declared as a global variable. The variable is visible to all functions and statements even if the functions are defined in different files. For

example, the following statement can use the variable **index1**:

```
index1 = index2 + 1;          (in file1.c)
```

However, global variables such as **index1** defined in one file are not visible to other files, unless they are also declared in other files. To do this, the storage keyword **extern** is used to declare external global variables. For example, when we want to define **index1** in *file1.c* to be available to another file *file2.c*, **index1** is declared as

```
extern int index1;          (in file2.c)
```

in *file2.c*. The keyword **extern** in *file2.c* declares that the real definition of **index1** is not in *file2.c* but is in another file. When the C compiler compiles *file2.c*, it will treat **index1** as if it has been defined in *file2.c*. Then, we can use **index1** in *file2.c* as follows:

```
k = index1;                  (in file2.c)
```

Therefore, the keyword **extern** may be used to specify the same variable to be used by different functions, in different files.

When we declare any variables with the **static** specifier *outside* functions, the variable is known as a static global variable. In the following statement:

```
static int index2;          (in file1.c)
```

the keyword **static** has special meaning. If a program consists of more than one file, a variable declared with the keyword **static**, e.g. **index2**, will make it invisible to other files. For example, **index2** is invisible to *file2.c*, but **index1** is visible to both *file1.c* and *file2.c*. A static variable is only visible in one file.

Static global variables make it possible for a few functions to share variables. This feature is very useful when developing large programs in which functions that need to share static global variables can be grouped into a file. With this, external variables that could cause undesirable side effects can be avoided.

Program 13.10 Storage classes for multiple-source-file programs.

```
/* file1.c */  
int index1;                /* global to all files */  
static int index2;         /* global to file 1 only */  
  
main(void)  
{
```

```

    int i;                                /* automatic variable */
    extern int fn(int, int); /* declare fn() to be available
                                from other files */

    for (i=0; i<10; i++) {
        int j;                            /* automatic variable */
        for (j=100; j<200; j++) {
            index2 = fn(i, j);
            index1 = index2 + 1;
            ....
        }
        ....
    }
    return 0;
}

/* file2.c */
extern int index1;                        /* index1 is defined in another
                                           file */

....
int fn(int i, int j)
{
    int k;
    static int m=0;                      /* declare m to be static */

    k = index1;
    m++;
    return m;
}

```

13.9 Storage Classes for Functions

A function, like a variable, can have storage classes. However, it can only be either external (the default) or static. An external function typically omits the keyword for the storage class **extern**. An external variable can be accessed by functions in other files. This is the same for **extern** functions:

```

double fn1();          /* extern is omitted */
extern int fn2();

```

A static function can be used only within the defining file:

```

static double fn3();

```

C can localize information and functions in a file by defining functions and global

variables as **static**. This is illustrated in Program 13.11. **fn()** has the program scope which allows the function to be used in other parts or files of the program. **fn3()** has the file scope, which makes the function only visible in this file, but not to other files.

Program 13.11 Localization of variables and function declarations.

```
int index1;           /* program scope */
static int index2;    /* file scope */
....
fn(....)             /* program scope */
{
    ....
}
static int fn3(...)   /* file scope */
{
    ....
}
```

13.10 ANSI C Type Qualifiers

Every C variable has two properties, i.e. the data type (such as **int** or **char**) and storage class (such as **auto**, **extern** or **static**). ANSI C adds one more property to variables called type qualifiers (**const** or **volatile**). The type qualifier **volatile** is used to indicate that a variable's storage can be altered by something besides the program statements that define the variable. This refers to hardware addresses and is mainly used in compiler optimization of variables. In this section, we will only discuss the **const** type qualifier.

The const Type Qualifier

The **const** keyword in a declaration establishes a variable whose value cannot be modified. The **const** variable must be initialized in its definition. Assignment statement, increment operation and decrement operation are not allowed.

```
const int x=7;  /* ok - initialization in definiton */
x = 12;        /* Error - assignment not allowed */
x++;          /* Error - increment operation not allowed */
x--;          /* Error - decrement operation not allowed */
```

A **const** array can be used to create an array of data that cannot be changed. It acts like an array of **const** variables. A **const** array must be initialized during declaration. No subsequent change is allowed.

```
const int scores[5] = { 70,54,34,23,65 };
scores[0] = 30;          /* Error */
scores[0]++;             /* Error */
scores[0]--;             /* Error */
```

The type qualifier **const** can also be used with pointers. There are two ways for this. The first is to define the pointer itself as **const**, and the second is to define the value which is pointed to as **const**. This can happen in the following three cases:

Case 1: `const char *ptr;`

This means that **ptr** points to a constant **char** value. Then, the pointed-to value (i.e. ***ptr**) cannot be changed but the value of **ptr** can be changed. Therefore, we have

```
*ptr = 'A';              /* Error */
ptr = "This is a pointer."; /* ok */
```

The first statement is an error, as **ptr** points to a **const** value. The second statement is valid. This is because **ptr** is a pointer variable that can be assigned to point to another string.

Case 2: `char *const ptr;`

This means that **ptr** is a **const** pointer. Then, the value of **ptr** cannot be changed but the pointed-to value (i.e. ***ptr**) can be changed. Therefore, we have

```
*ptr = 'A';              /* ok */
ptr = "This is a pointer."; /* Error */
```

Case 3: `const char *const ptr;`

It means that both **ptr** and ***ptr** cannot be changed.

The **const** keyword can also be used in formal function parameters. For example, consider the **strlen()** function definition:

```
int strlen(const char *str);
```

The function receives a pointer pointed to the beginning of the string. The function then returns the length of the string. The **strlen()** declaration implies that the string pointed to by **str** cannot be changed, but string pointer **str** can be changed. Therefore, we have

```
str++;          /* ok */
*str = 'A';     /* Error */
```

In another example:

```
char *strcat(char *, const char *);
```

the `strcat()` function appends the second string to the end of the first string. The first argument is a pointer to the first string that can be changed and updated. The second argument is a pointer to the second constant string that cannot be changed.

13.11 Exercises

1. Explain what is meant by the scope, linkage and storage duration of a variable.
2. Explain the meaning of the following types of scope: file scope, block scope and function prototype scope.
3. Explain the difference between global variable and local variable. Discuss what is bad programming practice regarding the use of local and global variables.
4. Give the outputs of the program:

```
#include <stdio.h>
int a=10;
static int p( );
int q( );

main()
{
    printf("The value of q( ) is : %d\n", q( ));
    printf("The value of p( ) is : %d\n", p( ));
    return 0;
}
static int p( )
{
    static int a=0;
    int b=10;
    a+= b;
    return a;
}
int q( )
{
```

```
    a+= p( );  
    a*= p( );  
    return a;  
}
```



The C Preprocessor

The C preprocessor processes a C source program before the compiler translates the program into object code. It provides several ways to modify a C program before the compilation process takes place. The preprocessor can replace symbolic abbreviations in programs with the directions specified using the **#define** preprocessor directive. It can also include other files in the program using the **#include** directive. In this chapter, we describe some of the features in the preprocessor.

This chapter covers the following topics:

- 14.1 The C Preprocessor
 - 14.2 The **#define** Directive
 - 14.3 Macros with Arguments
 - 14.4 File Inclusion
 - 14.5 Conditional Compilation
 - 14.6 Predefined Preprocessor Macros
 - 14.7 Other Directives and Operators
-

14.1 The C Preprocessor

Before discussing the purpose of the C preprocessor, let's consider the program shown in Program 14.1. In the program, the constant 6.0 is used repeatedly to represent the *average height* of a person and the *average daily salary* in a country. Two problems occur in the program. First, the program is difficult to read and understand, as it is not clear the real representation of the value 6.0, which can be used to represent the average height or average daily salary. In addition, it is also difficult to modify the program. To further improve the readability of this

program, Program 14.2 shows a modified version using the **#define** preprocessor directive which defines the constant value of 6.0 with constant names called **AVG_HEIGHT** and **AVG_SALARY**. The constant names are then used in the program. The use of symbolic constants will improve the clarity and portability of the program. And the program is easier to understand and modify.

Program 14.1 An example program.

```
main(void)
{
    float num1, num2, num3;
    num1 = 6.0;
    num2 = 100 * 6.0;
    num3 = 200 * 6.0;
    ....
    num3 = 6.0 * num1;
    ....
    return 0;
}
```

The C preprocessor is written to provide support for programming features not supported by the C compiler. In this case, it replaces every occurrence of **AVG_HEIGHT** and **AVG_SALARY** by 6.0 in the program before passing the program code to the compiler. After preprocessing the program code in Program 14.2, the program code shown in Program 14.1 is then obtained.

Program 14.2 An example using the #define directive.

```
#define AVG_HEIGHT 6.0
#define AVG_SALARY 6.0

main(void)
{
    float num1, num2, num3;
    num1 = AVG_HEIGHT;
    num2 = 100 * AVG_HEIGHT;
    num3 = 200 * AVG_SALARY;
    ....
    num3 = AVG_SALARY * num1;
    ....
    return 0;
}
```

As can be seen from the above example, the C preprocessor provides another convenient way in handling C programming. It operates on a one-line-at-a-time basis. Each line must start with the symbol '#'. Each '#' is followed by an identifier that is a command or directive name (e.g. **#define**, **#include**, etc.). If a directive spans more than one line, each line must be ended with the symbol '\'. Unlike C statements, preprocessor statements do not have a semicolon at the end of the statement. In addition, preprocessor statements cannot use variables or perform arithmetic operations as in C statements.

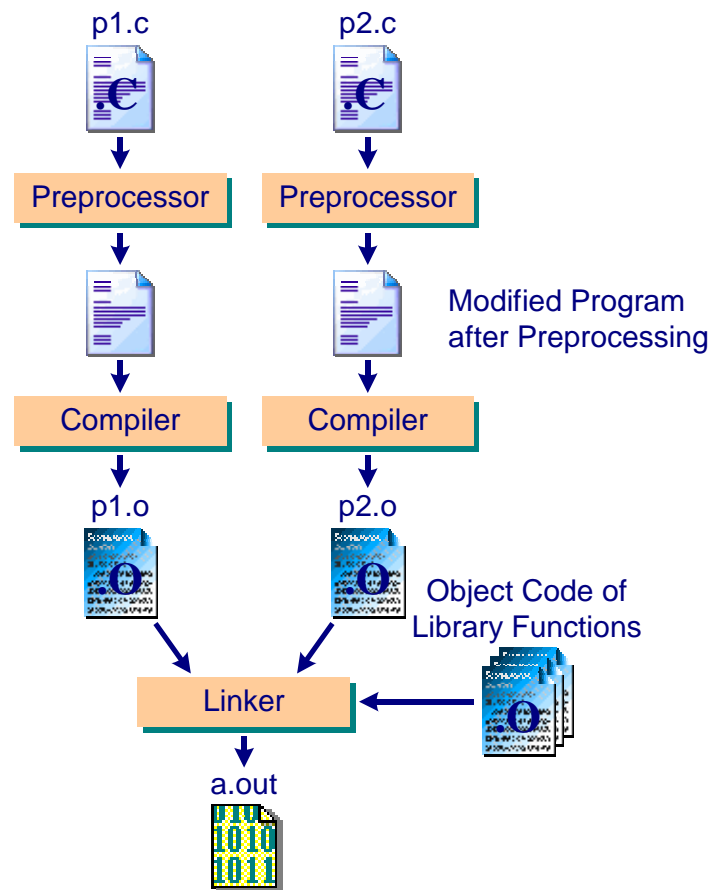


Figure 14.1 Program compilation process.

The C preprocessor and the C compiler are two different programs accepting different languages. The C preprocessor complements the C compiler to provide features not available in a C compiler. Figure 14.1 shows the program compilation process for a multiple-source-file program. The program consists of two files: *p1.c* and *p2.c*. They are compiled using a compiler. Before the compilation takes place, each source file goes through the preprocessor. The output of the preprocessor is

the source program with all the directions specified by the preprocessor directives carried out. This output file is then compiled and object code is generated for each file. The object code, *p1.o* and *p2.o* are linked with the object code from the C library to form the executable code. Which object code of the C library is used depends on the C library that is included in the source program. If no name is specified for the executable program, the executable code is stored in *a.out*.

14.2 The #define Directive

The **#define** preprocessor directive can appear anywhere in the source file, and the definition holds from the place where it occurs to the end of the file. Each **#define** line has 3 parts as shown in Figure 14.2: preprocessor directive, macroName and macroBody. When a macroName appears outside of its definition (referred to as *invocation*), it is replaced with its macroBody. This is called *macro expansion*. The replacement does not operate within strings (characters that appear between two double quotes). For example, the statement

```
printf("The AVG_HEIGHT is %d", AVG_HEIGHT);
```

will become

```
printf("The AVG_HEIGHT is %d", 6.0);
```

after macro expansion.

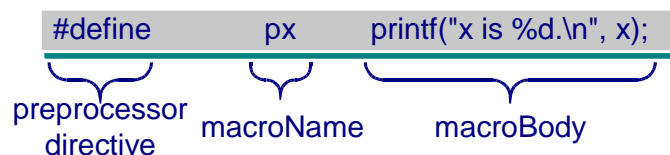


Figure 14.2 The #define directive.

A macro body that contains the names of other macros will cause repeated macro expansion. An example is shown in Program 14.3, which contains the program code before preprocessing.

Program 14.3 Program with macro definitions before expansion.

```
#include <stdio.h> /* stdio.h will be inserted here */  
#define COUNT1 10 /* comment is allowed here */
```



```
#define COUNT2 20 /* macroName is usually in capital
                  letters */
#define PROGRAMMING "Welcome \
to C programming"
/* a backslash for continuation of a definition to a new
   line */
#define TOTAL COUNT1+COUNT1
#define TOTAL2 TOTAL*COUNT2
#define PRINTMSG printf("The value is %d.\n", num);
#define MSG "The value is %d.\n"

main(void)
{
    int num = COUNT1;
    num = TOTAL2;
    PRINTMSG;
    printf(MSG, num);
    printf("%s\n", PROGRAMMING);
    printf("TOTAL:PROGRAMMING\n");
    return 0;
}
```

Program output

```
The value is 210.
The value is 210.
Welcome to C programming
TOTAL:PROGRAMMING
```

In Program 14.3, the statement

```
int num = COUNT1;
```

will expand and become

```
int num = 10;
```

as `COUNT1` is substituted by 10. The statement

```
num = TOTAL2;
```

becomes

```
num = TOTAL*COUNT2;
=> num = COUNT1+COUNT1*COUNT2;
```

```
=>  num = 10+10*20;
```

As the multiplication operator has higher precedence than the addition operator, it becomes

```
    num = 10+(10*20);  
=>  num = 210;
```

This might not be the intention of the program design. Therefore, to avoid this type of problem, we should enclose the macro body with brackets as follows:

```
#define TOTAL (COUNT1+COUNT1)
```

In this way, `num` will become

```
    num = (10+10)*20;  
=>  num = 400;
```

Similarly, we should also use parentheses for `TOTAL2` as follows:

```
#define TOTAL2 (TOTAL*COUNT2)
```

For the statements

```
PRINTMSG;  
printf(MSG, num);
```

they become the same statement after the expansion, which is

```
printf("The value is %d.\n", num);
```

The program after macro expansion is shown in Program 14.4.

Program 14.4 Program after macro expansion.

```
/* stdio.h will be inserted here */  
....  
main(void)  
{  
    int num = 10;  
    num = 10+10*20;  
    printf("The value is %d.\n", num);  
    printf("The value is %d.\n", num);  
    printf("%s\n", "Welcome to C programming");  
    printf("TOTAL:PROGRAMMING\n");  
}
```

```
    return 0;  
}
```

The replacement string is treated as a string of tokens (or words), rather than a string of characters. The macro definition

```
#define TOTAL COUNT1+COUNT1
```

has only one token, while the macro definition

```
#define TOTAL COUNT1 + COUNT1
```

has three tokens, i.e. `COUNT1`, `+` and `COUNT1`.

Consider the following macro definition:

```
#define TOTAL COUNT1      +      COUNT1
```

The preprocessor will treat the macro body with three tokens separated by a single space. So it is equivalent to

```
#define TOTAL COUNT1 + COUNT1
```

It is illegal to redefine a macro using a second `#define` statement. However, it is legal to repeat an identical `#define` statement. For example, the following macro definitions

```
#define TOTAL COUNT1      +      COUNT1  
#define TOTAL COUNT1 + COUNT1
```

which have the same three tokens are considered legal. But the macro definition

```
#define TOTAL COUNT1+COUNT1
```

has only one token, which is considered as redefining the macro `TOTAL`. Hence, this is illegal.

14.3 Macros with Arguments

A macro with arguments looks very much like a function, but it is not exactly the same as a function. The format for macros with arguments is

```
#define macroName(arg1, arg2, ...) macroBody
```

In Program 14.5, the function `sumOf2Number(int, int)` can be defined using the macro

```
#define SUMOF2NUMBER(i,j) (i) + (j)
```

When the preprocessor processes the source code, it replaces the macro in the statement `k = SUMOF2NUMBER(i, j);` with the macroBody and at the same time, it replaces the formal arguments with the actual arguments. Thus, the statement that is passed to the compiler will be

```
k = (i) + (j);
```

The value 30 is then assigned to the variable `k` when the program is executed.

Program 14.5 Macro with arguments.

```
#define SUMOF2NUMBER(i,j) (i) + (j)
main(void)
{
    int i,j,k;
    i = 10;
    j = 20;
    k = SUMOF2NUMBER(i, j);
    ....
}
```

The rules in defining a macro are that there must be no space between macroName and the first left parenthesis. Otherwise, `(arg1, arg2, ...)` is seen as part of the macroBody. Most compilers expect macros and preprocessor statements to be placed in the first column in the program. All formal arguments appearing in macroBody should always be parenthesized. Otherwise, unexpected results may occur. This is illustrated as follows. Consider the following macro definition:

```
#define SQUARE(x) x * x
```

which computes the square of the argument `x`. The statement

```
k = SQUARE(i);
```

will expand it to

```
k = i * i;
```

which is correct. However, the statement

```
k = SQUARE(i+j);
```

will expand it to

```
    k = i + j * i + j;  
=>  k = i + (j * i) + j;
```

The result is not what one expects as

```
k = (i + j) * (i + j);
```

If we define the square macro as

```
#define SQUARE(x)    (x * x)
```

The statement

```
k = SQUARE(i+j);
```

will expand to

```
k = (i + (j * i) + j);
```

which also has the same problem.

Therefore, to avoid this type of error, we can use as many parentheses as necessary. In this example, we can enclose each occurrence of the argument within parentheses and also enclose the macro with parentheses. Hence, if we define the square macro as

```
#define SQUARE(x)    ((x) * (x))
```

then the expansion becomes

```
k = ((i + j) * (i + j));
```

Another example is given on **SWAPINT** which swaps 2 integers. The macro is defined as

```
#define SWAPINT(x,y) { int temp=(x); (x)=(y); (y)=temp; }
```

We can call the macro in the **main()** function as

```
main(void)
{
    SWAPINT(low, high);
    ....
}
```

After expansion, the program code is

```
main(void)
{
    { int temp=(low); (low)=(high); (high)=temp; }
    ....
}
```

We can also perform a conditional test of leap year. The macro is defined as

```
#define ISLEAP(x)    (x)%4==0 && (x)%100!=0 || (x)%400==0
```

When the macro is called inside the `main()` function:

```
main(void)
{
    if (ISLEAP(year))
        ....
}
```

the program code is

```
main(void)
{
    if ((year)%4==0 && (year)%100!=0 || (year)%400==0)
        ....
}
```

after expansion.

A number of C functions have been written as macros. The macro names use lowercase letters. For example, in *stdio.h*, the macro **getchar** is defined as

```
#define getchar() fgetc(stdin)
```

This function **fgetc()** reads a single character from standard input, **stdin**. **putchar()** is also implemented as a macro.

Comparison of Macro and Function

Macros and functions are similar in that they both enable a set of operations to be represented by a single name. Since macro arguments are not variables, they have no types nor storage allocated to them. Their names do not conflict with variables of the same names. Generally, macros execute faster than functions because they have no function overheads such as saving variable values before function execution. Once defined, a macro name retains its meaning until the end of the source file or explicitly be removed with the **#undef** directive/command as

```
#undef macroName
```

However, we can use a pointer to point to a function but not to a macro because a macro is only recognized by the C preprocessor, but not the C compiler.

The advantages of macros compared to functions are that macros are usually faster than functions, and no type restriction is placed on arguments so that one macro may serve many data types. For example, the macro

```
#define ABS(x) ((x) < 0 ? -(x) : (x))
```

can be applied to data types **int**, **long**, **float** or **double** to give the absolute value.

However, there are several disadvantages with macros as compared to functions:

- Macro arguments are re-evaluated at each occurrence in the macro body, which leads to unexpected behavior if an argument contains side effects. For example, in the macro

```
#define min(i,j) ((i)>(j))? (j):(i)  
....  
a = min(b++, c);
```

it becomes

```
a = ((b++) > (c))?(c):(b++);
```

after expansion. If **b** is less than **c**, it gets incremented twice. Obviously, it is not the intention of the code. Therefore, it is recommended that we do not use side-effect operators such as **++** and **--** in a macro invocation.

- Function bodies are compiled once so that multiple calls to the same function can share the same code with no repetition. Macros are substituted with the macro bodies each time they appear in the program. The result is that a program with many large macros may be longer than the one that uses

functions instead. The general rule is that macros should be used for simple expressions. If the macro gets too large, then we should try to write it as a function.

- Macros do not check argument types. In contrast, compilers check ANSI function prototypes for both numbers of arguments and types.
- It is more difficult to debug programs that contain macros because of the additional layer of translation, as the program after macro expansion is not available for programmers to check.

14.4 File Inclusion

When the preprocessor detects an **#include** directive, it looks for the following filename and includes the content of that file within the current file. For example, if we have the following statements

```
#include "myAnyFile"
main(void)
{
    ....
}
```

and the **#include** file *myAnyFile* contains the following content:

```
#define True 1
#define False 0
```

then after preprocessing, the program replaces the **#include** line by the specified file. The statements become

```
#define True 1
#define False 0
main(void)
{
    ....
}
```

It is a convenient way to include other files, especially the header file (i.e. files with suffix **.h**), in a file. There are two formats for file inclusion. The first one is

```
#include <libraryFile>
```

For example, **#include <stdio.h>** causes the searching of the file *stdio.h* under the directory */usr/include* in a Unix system. Another form is


```
#include "myAnyFile"
```

This causes searching of *myAnyFile* from the user's directory, i.e. in the same directory as the file containing this particular **#include** directive.

We may also use **#include** directive to include a source program, but this is considered as a bad programming style. In addition, we may also make one include file to include another file. That is, the **#include** directive can be nested. Sometimes, if we need to have multiple include files for a program, this is considered a good method to place all the include directives within one file, and include this file in the program.

Header Files

At the beginning of each program file, we usually need to include the following information:

- The **#include** preprocessor directives, such as

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

- Macro definitions, such as

```
#define TRUE      1
#define FALSE     0
```

- Variables and functions defined in other files, such as

```
extern int variable1;
extern int fn2(int, int);
```

- New data type declaration, such as

```
typedef
```

If a program is divided into 10 files, all this information needs to be duplicated 10 times. To avoid typing the repeated information, we can place all this information into a header file, e.g. *myHeader.h*, and then include this header file in all the files using the **#include** preprocessor directive as follows:

```
#include "myHeader.h"
```

The definitions contained in the header file can be changed without the need to change all the files containing these definitions. However, all the files that include

the header file need to be recompiled.

The header file is different from that of the C standard library. The user header file is enclosed by double quotes (" ") while the C standard library header files are enclosed by arrows (< >). The C compiler knows where to look for the C library header files but not the user defined header files. By default, C compiler will look for the user defined header files in the directory where the user issues the compiling command. We can change the **#include** directive to contain an absolute address as follows:

```
#include "/home/staff/asschui/test.h"
```

14.5 Conditional Compilation

Conditional compilation uses the following directives: **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif** and **#endif**. It aims to selectively incorporate or omit a series of statements in a program. We can use conditional compilation to help placing debugging statements during program development and testing. It can also be used for improving program portability and handling header files.

The **#ifdef**, **#else** and **#endif** Directives

The **#ifdef**, **#else** and **#endif** directives are used in the following format:

```
#ifdef macroname
    statements will be included if macroName has been
    defined
#else
    statements will be included if macroName has not been
    defined
#endif
```

We can use conditional compilation to debug programming code as shown in Program 14.6. The program defines a constant called **DEBUG** using the **#define** statement. The **#ifdef** statement is used to test whether the macro name **DEBUG** is defined. If the macro has been defined, then the two **printf()** statements will be executed in the **main()** function. In the function **fn()**, the two **printf()** statements will also be executed when the macro **DEBUG** is defined. If **DEBUG** is not defined, then the **printf()** statement in the **#else** part will be executed.

Program 14.6 Conditional compilation for debugging.

```
#define DEBUG
main(void)
```

```
{
#ifdef DEBUG
    printf("debug statement1\n");
    printf("var1 = %d var2 = %d\n", var1, var2);
#endif
    ....
}
void fn()
{
#ifdef DEBUG
    printf("debug statement2\n");
    printf("var1 = %d var2 = %d\n", var1, var2);
#else
    printf("No debugging statement\n");
#endif
    ....
}
```

Preprocessing output: when the macro DEBUG is defined.

```
main(void)
{
    printf("debug statement1\n");
    printf("var1=%d var2=%d\n", var1,  var2);
    ....
}
void fn()
{
    printf("debug statement2\n");
    printf("var1=%d var2=%d\n", var1,  var2);
    ....
}
```

Preprocessing output: when the macro DEBUG is not defined.

```
main(void)
{
    ....
}
void fn()
{
    printf("No debugging statement\n");
    ....
}
```

The #if, #else and #elif Directives

The format for **#if** directive is as follows:

```
#if constant_expression1
    statementList1
#elif constant_expression2
    statementList2
#else
    statementList3
#endif
```

The **#if** directive works in a very similar way to the **if** statement. Only constant expressions can be used with **#if**, and variables are not allowed. In addition, we can also use relational and logical operators such as

```
#define DEBUG 1
#if DEBUG == 1
....

#define COUNT 10
#if COUNT > 10
....
```

Program 14.7 illustrates the use of **#if** statement in conditional compilation for debugging. If **DEBUG** is true (i.e. nonzero), then the two **printf()** statements enclosed by **#if** will be executed. If **DEBUG** is false (i.e. 0), then the **printf()** statement inside the **#else** part is executed.

Program 14.7 Conditional compilation using #if statement.

```
#define DEBUG 1
main(void)
{
    #if DEBUG
        printf("debug statement1\n");
        printf("var1=%d var2=%d\n", var1, var2);
    #endif
    ....
}
void fn()
{
    #if DEBUG
        printf("debug statement2\n");
        printf("var1=%d var2=%d\n", var1, var2);
```

```
#else
    printf("No debugging statement\n");
#endif
    ....
}
```

The preprocessor also recognizes **#if defined**, which has the following format:

```
#if defined(macroName)
```

which has the equivalent meaning of

```
#ifdef macroName
```

The **#if**, **#elif** and **#else** directives can also be used to customize program code for writing programs targeted for different hardware and operating environments. For example, in the following macros:

```
#if defined(SUN)
#include "sun.h"
#elif defined(IBM)
#include "ibm.h"
#elif defined(BORLAND)
#include "borland.h"
#else
#include "generic.h"
#endif
```

if the macro **SUN** is defined, the header file "**sun.h**" is included, else if the macro **IBM** is defined, the header file "**ibm.h**" is included, etc. The header file "**generic.h**" will be included if all the specified macros have not been defined. By using the **#if**, **#elif** and **#else** directives, different test conditions can be evaluated in order to include the appropriate header file under a specific environment.

The #ifndef Directive

The **#ifndef** directive opposes the **#ifdef** directive. It works with the **#endif** directive. The format for the **#ifndef** and **#endif** directives is:

```
#ifndef macroname
    statements will be included if macroname is not defined
#endif
```

The `#ifndef` directive can be used to ensure that a header file is included only once in a program. It is because the compiler will give out an error message if the same statements are included more than once. Therefore, there is a need to ensure that the statements are included only once. For example, in the program `myProgram.c`, we have the following code:

```
#include    "userHeader1.h"
#include    "userHeader2.h"
main(void)
{
    ....
}
```

The contents for the header files are:

```
/* This is userHeader1.h */
#include    <stdio.h>
.....

/* This is userHeader2.h */
#include    <stdio.h>
.....
```

Since `myProgram.c` includes `userHeader1.h` and `userHeader2.h`, `stdio.h` is indirectly included twice in `myProgram.c`. To tackle this problem, we can use the `#ifndef` directive in the `stdio.h` file as

```
/* This is stdio.h */
#ifndef STDIO
#define STDIO
    ....          /* the original contents of stdio.h */
#endif
```

where a local macro name `STDIO` is defined to check whether the `stdio.h` has already been included. In fact, all the header files, e.g. `userHeader1.h` and `userHeader2.h` should put in place the similar test as follows:

```
/* This is userHeader1.h */
#ifndef USERHEADER1 /* placed at the beginning of the
                    file*/
#define USERHEADER1
    ....
#endif              /* placed at the end of the file */
```

```

/* This is userHeader2.h */
#ifndef USERHEADER2
#define USERHEADER2
    ....
#endif

```

The **#ifndef** and **#define** are placed at the beginning of the header files, and the **#endif** statement is placed at the end of the header file. If the macro on header file (i.e. **USERHEADER1** or **USERHEADER2**) has been defined, then the preprocessor ignores all the statements in the file until the **#endif** statement. If the macro is not yet defined, then all the statements in the file will be processed.

The #undef Directive

For example, if we have defined

```
#define COUNT 10
```

the **#undef** directive

```
#undef COUNT
```

cancels any definitions that have been defined before. Therefore, if we are not sure whether a particular name has been used before, we should undefine it first. It is still valid to use **#undef**, even if the name has not been defined before.

14.6 Predefined Preprocessor Macros

ANSI C has 5 predefined macros which are given in Table 14.1. They cannot be undefined by the programmer. Each of the macro names includes two leading and two trailing underscore (__) characters.

Table 14.1 Predefined preprocessor macros.

Macro	Meaning
__DATE__	a string containing the current date the source file is compiled (in the format "mm/dd/yyyy")
__FILE__	a string containing the current source filename
__LINE__	an integer representing the current line number of the source code
__STDC__	the integer constant 1. This is intended to indicate that the implementation is ANSI compliant

`__TIME__` a string containing the current time the source file is compiled (in the format "hh:mm:ss")

14.7 Other Directives and Operators

The format for the `#line` directive is

```
#line lineNumber "fileName"
```

The directive causes the compiler to renumber the source text so that the next line is numbered as `lineNumber` and the current file name is `fileName`. The `fileName` may be omitted. For example, the directive

```
#line 1
```

starts the line number from 1 beginning with the following line, and the next line as line 2, etc. If a `fileName` is included, the directive

```
#line 1 "file.c"
```

specifies that the following line is numbered as 1, and so on, and associates any compiler messages with the file `"file.c"`. Compiler messages include messages generated by syntax errors and compiler warnings.

The format for the `#error` directive is

```
#error stringMessage
```

The directive is used to generate preprocessing time error messages. For example, when the `#error` directive

```
#if A_SIZE < B_SIZE
#error "Incompatible sizes"
#endif
```

is processed, the compiler generates the error and displays the error message. The program does not compile. This allows programmers to add their own error-reporting capabilities.

The `#pragma` directive is used to specify arbitrarily defined directives. Each different C compiler will have a different set of pragmas. See the documentation in the C implementation for more detail.

The `#` is the unary operator that causes the formal arguments in a macro definition to be surrounded by double quotes. For example, consider the following

macro definition:

```
#define errorMessage(a, b) printf(#a #b "\n")
```

When we call the statement

```
errorMessage(Error:, this is error 11);
```

we have

```
printf("Error:" "this is error 11" "\n");
```

after macro expansion. Strings separated by white spaces are then concatenated. Thus, it is equivalent to

```
printf("Error:this is error 11\n");
```

The `##` is a binary operator which merges its two operands. For example, consider the following directive

```
#define X(a,i) a##i
```

When we call the statement

```
X(1,5);
```

it concatenates its two arguments and replaces `X(1,5)` by `15` in the program.

14.8 Exercises

1. State the advantages and disadvantages of using macros as compared to using functions.
2. Explain what is meant by "inline expansion". Assume that the following macro has been defined:

```
#define double(x) ((x)*2)
```

Perform the inline expansion for the following statement:

```
b = double(a*a);
```

3. Consider the following header file.

```
1  #ifndef HEADERFILE
2  #define HEADERFILE
3
4  #define DEBUG 1
5
6  #include <stdio.h>
7  #include "anotherfile.h"
8
9  #define power(x,n,y)
10     {y=1; for (i=1; i<n; i++) { y = y * x; } }
11
12 #endif
```

- (a) Explain the meanings of lines 1, 4, 6 and 7 in the header file.
- (b) Lines 9 and 10 contain a macro definition for a power function which computes x^n and assigns the result to y. Identify the errors in this macro definition.

4. The power function can be defined recursively as

$$\begin{aligned} x^n &= x * x^{n-1} \text{ for } n \geq 1 \text{ and } n \in \text{integer}; \text{ and} \\ x^0 &= 1 \quad \text{for } n=0. \end{aligned}$$

Determine if the recursive version of the power function can be implemented using a macro definition. If yes, write out the macro definition. Otherwise, state why it cannot be implemented using a macro.

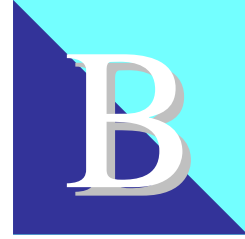


ASCII Character Set

The following table lists the American Standard Code for Information Exchange (ASCII) character codes in decimal values. The ASCII code is a 7-bit code which ranges from 0 to 127.

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

Character codes 0-31 and 127 are non-printable characters. Each character in the table is read in row-column order, e.g. 'A' has a decimal value of 65, and 'a' has a decimal value of 97.



Answers to Exercises

Chapter 1: Computer Systems and C Programming

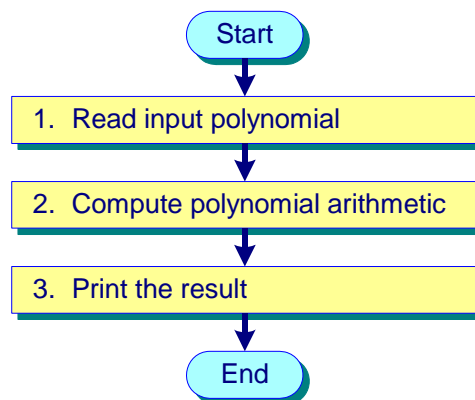
1. The four basic components of a computer hardware system are the central processing unit (CPU), main storage, secondary storage and input and output (I/O) devices. CPU controls and coordinates the operations of the system. Main memory is to store information that is to be processed by the CPU. Secondary storage is needed to store information that requires to be kept for a long period of time. Input and output devices support the communication with the computer system.
2. Application software refers to programs that are used to solve specific problems. System software refers to programs that are used by programmers or developers to build application software.
3. Machine language consists of instructions in binary (or hexadecimal) code. Assembly language replaces binary instructions by symbolic instructions in text. High-level languages allow programmers to write programs closer to English.
4. The five high-level programming languages include PASCAL, BASIC, C, COBOL and Prolog.
5. The strengths of C language include flexibility, efficiency, portability and modularity. The weaknesses of C programming include the free style of expression in C that can make it difficult to read and understand. As C is not a strongly-typed language, it is the programmer's responsibility for ensuring the

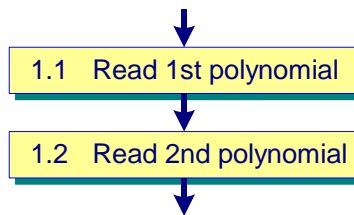
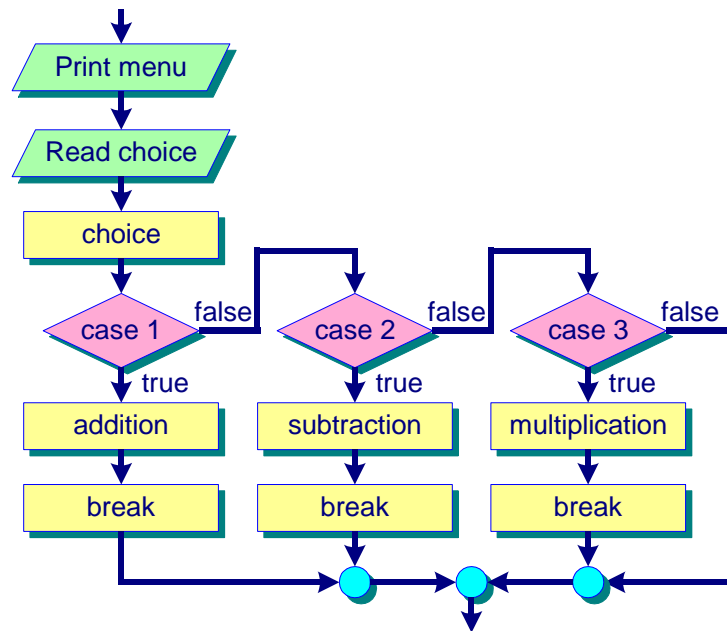
correctness of the program. Pointer in C is a very useful feature, however, it can also cause programming errors that are difficult to debug and trace.

6. The program will go through the preprocessor, compiler and linker for preprocessing, compilation and linking processes. Preprocessor preprocesses the directives such as `#include` and `#define`. The input is `prog.c`. The output replaces the macros by text files as indicated by `#include`. Compiler translates the output from preprocessor into machine code. The input is the output from the preprocessor. The output is the machine code file. Linker links the machine code with other compiled code like library routines to produce the final executable program of `prog.c`. The input is the output from the compiler. The output is `prog`.

Chapter 2: Program Development

1. The six steps in program development process are problem definition, problem analysis, program design, implementation, program testing and documentation.
2. An algorithm consists of a series of step-by-step instructions for the solution of a problem. Flowcharts and pseudocode are two commonly used methods for describing algorithms. Flowcharts are pictorial description of algorithms, while pseudocode is an informal language that is used for describing algorithms.
3. Top-down stepwise refinement is a sequence of refinement steps to break down a problem into subproblems and subproblems into even more simpler problems. With each refinement, more details of the algorithm is designed. The process repeats until enough details are designed to solve the problem.
- 4.



Refinement for 1:Refinement for 2:**Chapter 3: Data Types, Constants and Variables**

1. (a) 3_persons - no, it is not allowed to start with a digit.
 (b) three persons - no, space is not allowed.
 (c) double - no, double is a keyword.
 (d) special-rate - no, the '-' is not allowed.
 (e) _wins - yes.
 (f) one_&_two - no, the '&' is not allowed.
 (g) period. - no, the period at the end is not allowed.
 (h) \$dollar - no, it is not allowed to start with the '\$' sign.
2. (a) '5' - character
 (b) 45 - decimal integer
 (c) 4.5 - floating point
 (d) '\045' - ASCII code in octal for the character '%'

- (e) 0x45 - hexadecimal integer
 - (f) '\n' - character (output: a newline)
 - (g) 123456789L - decimal long integer
 - (h) 4.567F - floating point
 - (i) -456 - negative decimal integer
 - (j) 0456 - octal integer (starts with '0')
 - (k) 0Xabc - hexadecimal integer
 - (l) 0xaBcDeF12L - hexadecimal long integer
3. (a) var is declared twice.
- (b) The value 1,25 should be 1.25 for the variable var.
- (c) The keyword integer data type should be **int**, not integer.
- (d) The keyword long is not permitted to be declared as a variable name.
- (e) \$discount is not a valid identifier. It cannot start with a '\$' character.
- (f) var-2 is not a valid identifier. It cannot include the '-' character.
4. The data type **int** occupies two bytes in memory in most computers, while data type **float** takes 4 bytes and data type **double** takes 8 bytes in most computers. For small integer values, data type **int** helps to save memory space over the data types **float** and **double**.

Chapter 4: Simple Input/Output

1. (a) The output is 4.
- (b) The output is $x + x =$.
- (c) The output is $x + x = 4$.
- (d) The output is $5 = 5$.
2. (a) The output is X 58 88 130.
- (b) In the output, abcd are printed, then '\r' will return the cursor to the position where a is printed. Subsequently, 1234 is printed which overwrites abcd. The '\b' moves the cursor back to the position of 4, and 5 is printed, which overwrites 4. Finally, the cursor goes to the first position of the next line.
- (f) The output is the five beeps from the terminal. Each of the output is the representation of the ASCII code 00000111.
3. (a) $f = 1.234500$ $g = 5.678900$
- (f) $f = 1.23$ $g = 5.67$
4. `scanf("%d/%d/%d", &day, &month, &year);.`

Chapter 5: Operators, Expressions And Assignments

1. (a) $(a + b) / (x + (c - d))$
 (b) $(a + b) / x * x * (c + d) * (c + d)$
2. (a) $i = 6$ (b) $i = 5$ (c) $i = 5$ (d) $i = 6$
3. (a) $i*j/j = 10*3/3 = 10$ (integer)
 (b) $i/j*j = 10/3*3 = 1$ (integer)
 (c) $(--i)*j = 9*3 = 27$ (integer)
 (d) $(i++)*j = 10*3 = 30$ (integer)
 (e) $(int)a*10 = int(1.2)*10 = 1*10 = 10$ (integer)
 (f) $int(a*10) = int(1.2*10) = int(12.0) = 12$ (integer)
 (g) $(i+a)*j = (10.0+1.2)*3 = 11.2*3.0 = 33.6$ (floating point)
 (g) $(i+j)*a = (10+3)*1.2 = 13.0*1.2 = 15.6$ (floating point)
4. ans1 is 1.0 and ans2 is 19.

Chapter 6: Branching

1. (a) $x \&\& y \&\& z = (x \&\& y) \&\& z = 1$
 (b) $x \parallel y \&\& z = x \parallel (y \&\& z) = 1$
 (c) $x \&\& y \parallel z = (x \&\& y) \parallel z = 1$ ($\&\&$ has higher precedence than \parallel)
 (d) $x \parallel !y \&\& z = x \parallel ((!y) \&\& z) = 1$ (precedence order: $!$, $\&\&$, \parallel)
 (e) $x-- \leq y$ is $(x--) \leq y$ gives 1
 (f) $x < y \&\& z$ is $(x < y) \&\& z$ gives 1
2. $(!(!x \&\& !y) \parallel z) = (!(!1 \&\& !2) \parallel 0) = (!(!true \&\& !true) \parallel false) = !(false \&\& false) \parallel false = (!false \parallel false) = (true \parallel false) = true$. Therefore, the output is First Statement.
3.

```
int b1,b2,s1,s2; int a1,a2,a3,a4;
if (a1 <= a2) { b1 = a2; s1= a1; }
else { b1 = a1; s1= a2; }
if (a3 <= a4) { b2 = a4; s2 = a3; }
else { b2 = a3; s2 = a4; }
if (b1 <= b2)
    printf("The biggest value is %d\n", b2);
else
    printf("The biggest value is %d\n", b1);
if (s1 <= s2)
    printf("The smallest value is %d\n", s1);
else
```



```
printf("The smallest value is %d\n", s2);
```

4. `#include <stdio.h>`

```
main(void)
{
    char ch;
    ch = getchar();
    if (ch > 64 && ch < 91)
        printf("The character is a upper case letter.\n");
    else if (ch > 96 && ch < 123)
        printf("The character is a lower case letter.\n");
    else if (ch > 47 && ch < 58)
        printf("The character is a numeral.\n");
    return 0;
}
```

5. `switch ((number+5)/10) {`

```
    case 10:
    case 9:
    case 8:  printf("%c\n", 'A');
             break;
    case 7:  printf("%c\n", 'B');
             break;
    case 6:  printf("%c\n", 'C');
             break;
    case 5:  printf("%c\n", 'D');
             break;
    default: printf("%c\n", 'F');
}
```

Chapter 7: Looping

1. `#include <stdio.h>`

```
main(void)
{
    int count, a, min, total=0;
    scanf("%d", &min);
    total = min;
    for (count = 0; count < 9; count++) {
        scanf("%d", &a);
        total += a;
        if (a < min)
            min = a;
    }
}
```

```

    printf("The total and the smallest value are %d %d\n",
        total, min);
    return 0;
}

```

2. #include <stdio.h>

```

main(void)
{
    int m,n,k,height;
    printf("Enter a number : ");
    scanf("%d", &height);
    for (m=0; m < height; m++)
    {
        for (k=0; k<(height - m -1); k++)
            putchar(' ');
        for (n=0; n<m+1; n++)
            printf("%d", height);
        printf("\n");
    }
    return 0;
}

```

3. #include <stdio.h>

```

main(void)
{
    int a, b, height, upper, lower, lines;
    printf("Please enter the height of the triangular
        pattern:");
    scanf("%d", &height);
    upper = height/2 +1;
    lower = height/2;
    /* print upper triangle */
    for (lines=1; lines<=upper; lines++) {
        for (a=1; a<=(upper - lines); a++)
            putchar(' ');
        for (b=1; b<= (2*lines - 1 ); b++)
            putchar('*');
        putchar('\n');
    }
    /* print lower triangle */
    for (lines=lower; lines>=1; lines--) {
        for (a=1; a<= (lower - lines + 1); a++)
            putchar(' ');
        for (b=1; b<= (2*lines - 1 ); b++)
            putchar('*');
    }
}

```

```

    putchar('\n');
}
return 0;
}

```

```

4. main()
{
    double x, temp, term;
    int n, sign=1, d1=1, d2=2;

    printf("Please enter the value of x\n");
    scanf("%lf", &x);

    term = temp = x;
    for (n=3; n<=19; n+=2) {
        sign = -sign;
        term *= x*x;
        temp += sign * (term*d1/(d2*n));
        d1 *= n;
        d2 *= n+1;
    }
    printf("result = %f\n", temp);
}

```

5. The break and continue statements are used to alter the flow of control in the while, for, do/while structures. The break statement causes immediate exit from the structure in which it is contained. It can be used with the switch statement. The continue statement skips the remaining statements in the body of the loop and performs the next iteration of the loop.
The values are $i = 10$ and $j = 12$.

6. The program outputs are:

```

x = 1, total = 1
x = 3, total = 4
x = 5, total = 9
x = 7, total = 16
x = 9, total = 25

```

The modified function is given as follows:

```

void g()
{
    int x=0, y=10, total=0;
    while (x < y) {

```

```

        x += 1;
        if (x % 2 != 0) {
            total += x;
            printf("x = %d and total = %d\n", x, total);
        }
    }
}

```

Chapter 8: Functions

1. In call by value, when a function is called, the value of a variable is passed to the function. This is used when the calling body needs to pass a value to the function. In call by reference, when a function is called, the address of a variable is passed to the function so that the function can write to the location where the address is provided. This is used to pass a value back to the calling body.
2. A recursive function either calls itself directly or indirectly. Recursion usually consists of a general case and one or more terminating conditions. It is vital in the program implementation that the recursive function can terminate through its terminating conditions. An example is given in the factorial of a positive number n ($n!$) where the general condition is $n! = n*(n-1)*(n-2)*... *2*1$, and the terminating condition is $0! = 1$. The advantages of recursive function are clear and short code. However, the disadvantages of recursive function are that they are expensive in terms of execution time and memory space. Therefore, a function should be implemented recursively and not iteratively when the problem itself is recursive in nature.

3.

```
void drawRectangle (int width, int length, char chr)
{
    int m,n;
    for (m=0; m < (width-1); m++)
    {
        for (n=0; n<(length-1); n++)
            putchar(chr);
        printf("\n");
    }
}
```

4. The outputs of the program are:

```

x = 10, y = 10, *z = 100
x = 10, y = 10, *z = 100
x = 12, y = 11, *z = 100

```

```
x = 14, y = 12, *z = 100
x = 16, y = 13, *z = 100
x = 18, y = 14, *z = 100
n1 = 20, n2 = 15, n3 = 100
```

5. (a)

```
int iDigits(int n)
{
    int count = 0;
    do {
        count ++;
        n = n/10;
    } while (n > 10);

    return count;
}
```
- (b)

```
void iDigits(int n, int *nd)
{
    *nd = 0;
    do {
        (*nd)++;
        n = n/10;
    } while (n > 0);
}
```
- (c)

```
int rDigits(int n)
{
    if (n < 10)
        return 1;
    else
        return rDigits (n/10) + 1;
}
```
- (d)

```
void rDigits(int n, int *nd)
{
    if (n < 10)
        *nd = 1;
    else {
        rDigits(n/10, nd);
        *nd = *nd + 1;
    }
}
```
6.

```
int digitvalue(int n, int k)
```

```

{
    int i;
    int r;
    for (i=0; i<k; i++)
    {
        r = n%10;
        n /= 10;
    }
    return (r);
}
int rdigitvalue(int n, int k)
{
    if (k==0) return 0;
    if (k==1) return n%10;
    return (rdigit(n/10, k -1));
}

```

7. void GroupDigits(long n, long *group1, long *group2)

```

{
    static long gp1power=1;
    static long gp2power = 1;
    static long gp1result=0;
    static long gp2result=0;
    long digit;
    digit = n % 10;
    if (digit < 5) {
        gp1result += digit * gp1power;
        gp1power *= 10;
    }
    else {
        gp2result += digit * gp2power;
        gp2power *= 10;
    }
    n /= 10;

    if (n<=0) {
        if (gp1power ==1) *group1 = -1;
        else *group1 = gp1result;
        if (gp2power ==1) *group2 = -1;
        else *group2 = gp2result;
        gp1result = 0; gp2result=0;
        gp1power = 1; gp2power=1;
    }
    else
        GroupDigits(n, group1, group2);
}

```

```
8. long gcd (long num1, long num2)
{
    long i, remainder;

    if (num1 < num2) { /* determine num1 is the larger
        number */
        i = num1;          /* otherwise swap */
        num1 = num2;
        num2 = i;
    }
    remainder = num1 % num2;
    if (remainder == 0)
        return (num2);
    return (gcd(num2, remainder));
}
```

Chapter 9: Array

```
1. (a) void ReadInput(void)
{
    int i,j;
    for (i=0; i<20; i++)
        for (j=0; j<5; j++)
            scanf ("%d ", &production[i][j]);
}

(b) float ComputeAverage(void)
{
    int i,j, total;

    total = 0;
    for (i=0; i<20; i++)
        for (j=0; j<5; j++)
            total += production[i][j];
    return (float(total)/20.0);
}

(c) void FindtheBest(void)
{
    int weekttotal[20];
    int i, j, highest;
    for (i=0; i<20; i++)
    {
        weekttotal[i] = 0;
```

```

        for (j=0; j<5; j++)
            weektotal[i] += production[i][j];
    }
    highest = -1;
    for (i=0; i<20; i++)
    {
        if (weektotal[i] > highest)
            highest = weektotal[i];
    }
    for (j=0; j<20; j++) {
        if (weektotal[j] == highest)
            printf("worker %d is the most productive\n", j);
    }
}

```

2. The purpose of the program is to compare the two input arrays, ar1 and ar2, element by element, and put the smaller one into a third array ar3. If the two input arrays are in ascending order, the resulting array ar3 will also be in ascending order. After applying the input data, ar3 becomes { -3, -1, 0, 1, 1, 2, 3, 5, 7, 7, 8, 9, 9, 11}.

3. `void reverseAr(int ar[], int size)`
- ```

{
 if (size > 0) {
 reverseAr(&ar[1], size-1);
 printf ("%d ", ar[0]);
 }
}

```

4. Iterative function:
- ```

int find(int size, int array[], int number)
{
    int j;
    for (j = 0; j < size; j++)
        if (array[j] == number) return j;
    return -1;
}

```

Recursive function:

```

int find(int size, int array[], int number)
{
    int j;
    if (size == 1)
        if (array[0] == number)

```



```

        return 0;
    else return -1;
else {
    j = find(size -1, &array[1], number);
    if (j == -1) return -1;
    else return j + 1;
}
}

```

Chapter 10: Character Strings and String Functions

1. A string is one-dimensional array of type char. The null character '\0' is used to delimit the string.
2. The first declaration is equivalent to the statement `char str1[] = "abcd";` `str1` and `str2` are address constants, while `str3` is a pointer variable.
 - (a) invalid; RHS is an address while the LHS is a const pointer.
 - (b) valid.
3. The function removes char from the string to form a new string. At the same time, the number of characters in the string is returned to the calling function. For example, if the string `str = "This is a string."` and `char = 's'`, then after executing the function, `str` becomes `"Thi i a tring"` and `z = 3` is returned.
4. If the input is STAR, then the output of `str` is RATS.
5. The unknown C function constructs the word formed by the first characters of the words in the string `str1`. The word constructed should be stored in the string `str2`. The string `str1` can be of any length, but it is always terminated with the NULL character. Similarly, the string `str2` must also be terminated with the NULL character. Moreover, any two words in `str1` are assumed to be separated by a space character. If the input string is `"How are you?"`, then the output string is `"Hay"`.

6.

```

char *locatechr(char *str, char ch)
{
    char *s;
    *s = NULL;
    while (*str != '\0') {
        if (*str == ch)    s = str;
        str++;
    }
    return (s);

```

```
}
```

```
7. #include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
    if (argc != 3)
        printf ("Input the following command: %s number
                number\n", argv[0]);
    else
        printf("%d\n", atoi(argv[1]) + atoi(argv[2]));
}
```

8. The outputs are:

```
&a[1][2] = hi
&a[1][4] =
&b[4] = 456789
c[0]+1 = bcdefghi
c[1]+2 = hi
ptr-3 = defghi
```

Chapter 11: Structures, Unions and Enumerated Types

1. Structure is an aggregate type that consists of a collection of subcomponents and treated as a single entity. Union has the same syntactic form as structured objects. However, subcomponents of union share the same storage overlaid upon each other. Therefore, using union can save memory space needed.
2. (a) Invalid – because the type of person1 is person rather than the character string.
 (b) Valid.
 (c) Invalid – because it is not possible to apply relational operators on structures.
 (d) Invalid – this can only be done by using strcpy().

```
3. #include <stdio.h>
typedef struct {
    char title[81];
    char lastname[81];
    char firstname[81];
    char publisher[81];
    int day, month, year;
} booktype;
```

```

main(void)
{
    booktype book;
    char repeat = 'y';
    do {
        printf("Enter the title of the book : ");
        gets(book.title);
        printf("Enter the author's first name : ");
        gets(book.firstname);
        printf("Enter the author's last name : ");
        gets(book.lastname);
        printf("Enter the date of publication as (dd-mm-yy)
            : ");
        scanf("%d-%d-%d", &book.day, &book.month,
            &book.year);
        fflush(stdin);
        printf("Enter the publisher name : ");
        gets(book.publisher);
        printf("\n");
        printf("Title of the book: %s\n", book.title);
        printf("Author of the book: %s %s\n",
            book.firstname, book.lastname);
        printf("Date of publication %d-%d-%d\n", book.day,
            book.month, book.year);
        printf("Publisher of the book: %s\n",
            book.publisher);
        printf("\n");
        printf("Continue ('y' or 'n') : ");
        scanf("%c", &repeat);
        fflush(stdin);
    }while (repeat == 'y');
}

```

4. `complex add_complex (complex c1, complex c2)`

```

{
    complex result;
    result.real = c1.real + c2.real;
    result.imag = c1.imag + c2.imag;
    return result;
}
complex sub_complex (complex *c1, complex *c2)
{
    complex result;
    result.real = c1->real - c2->real;
    result.imag = c1->imag - c2->imag;
    return result;
}

```

```

}
complex mul_complex (complex c1, complex c2)
{
    complex result;
    result.real = c1.real * c2.real - c1.imag * c2.imag;
    result.imag = c1.imag * c2.real + c1.real * c2.imag;
    return result;
}
complex div_complex (complex *c1, complex *c2)
{
    complex result;
    double den;
    den = c2->real * c2->real + c2->imag * c2->imag;
    if (den != 0.0) {
        result.real = (c1->real * c2->real + c1->imag * c2->imag) / den;
        result.imag = (c1->imag * c2->real - c1->real * c2->imag) / den;
        return result;
    }
    else
        printf ("error !");
}

```

The code for the whole program:

```

#include <stdio.h>
complex add_complex(complex c1, complex c2);
complex mul_complex(complex c1, complex c2);
complex sub_complex(complex *c1, complex *c2);
complex div_complex(complex *c1, complex *c2);

main(void)
{
    char choice;
    complex input1, input2, result;
    do {
        printf("a - addition \n");           /* print menu */
        printf("s - subtraction \n");
        printf("m - multiplication \n");
        printf("d - division \n");
        printf("q - quit \n");
        printf("Enter your choice : ");       /* get choice */
        scanf("%c", &choice);
        if (choice == 'q') exit(0);          /* input */
    }
}

```

```

printf("Enter your complex number 1 (format: real
      <space> imag) : ");
scanf("%f %f", &input1.real, &input1.imag);
printf("Enter your complex number 2 (format: real
      <space> imag): ");
scanf("%f %f", &input2.real, &input2.imag);

switch (choice) {          /* arithmetic operations */
    case 'a':result = add_complex(input1, input2);
                break;
    case 's':result = sub_complex(&input1, &input2);
                break;
    case 'm':result = mul_complex(input1, input2);
                break;
    case 'd':result = div_complex(&input1, &input2);
                break;
}
printf("%f %f", result.real, result.imag);
    /* output */
}while (1);
}

```

Chapter 12: File Input/Output

1. A stream is a sequence of characters that connects a program to a device (or file) and transfers the data in (input stream) and out (output stream) from the program. Three standard streams are predefined in C. They are also known as standard input/output files. The standard streams are referred by three **FILE** pointers **stdin** (standard input), **stdout** (standard output) and **stderr** (standard error). These streams are defined in the standard I/O header file *stdio.h*. A C program normally uses the standard streams **stdin** and **stdout** as its sources for input and output. Stream or file redirection allows a program designed to use keyboard and display to redirect the input and output to and from files. This has the advantage that the program that uses functions such as **getchar()**, **scanf()**, **putchar()** and **printf()** can be run with file input and output without the need for modification.

2.

```
#include <stdio.h>
typedef struct {
    char id[10];
    int lab[5];
    int exam;
} studentRec;
studentRec stud;
```

```

void main()
{
    FILE *filedat;
    int i, sum;
    float    finalscore;
    if ((filedat=fopen("sc103.dat","r")) == NULL) {
        fprintf(stderr, "File cannot be opened!\n");
        exit();
    }
    fscanf(filedat, "%7s", stud.id);
    while (!feof(filedat)) {
        sum = 0;
        for (i=0; i< 10; i++) {
            fscanf(filedat, "%d", &stud.lab[i]);
            sum += stud.lab[i];
        }
        fscanf(filedat, "%d", &stud.exam);
        finalscore = sum*0.2/10 + stud.exam*0.8;
        printf("%s : %f\n", stud.id, finalscore);
        fscanf(filedat, "%7s", stud.id);
    }
    fclose (filedat);
}

```

3. #include <stdio.h>

```

int main(int argc, char *argv[ ])
{
    int i, n;
    char str[100];
    FILE *fp;

    if (argc != 3) {
        fprintf (stderr, "Usage: %s  n  filename\n",
            argv[0]);
        return (0);
    }
    fp = fopen (argv[2], "r");
    n = atoi(argv[1]);
    for (i=0; i<n; i++) {
        fgets(str, 100, fp);
        printf("%s", str);
    }
    fclose (fp);
    return 0;
}

```

```

4. #include <stdio.h>
   #include <ctype.h>
   #define MAX 100 /* assume a maximum of 100 characters per
   line */
   int main(int argc, char *argv[ ])
   {
       int i=0;
       char str[MAX];
       FILE *fp;

       if (argc != 2) {
           fprintf (stderr, "Usage: %s filename\n", argv[0]);
           return (0);
       }
       if ((fp = fopen (argv[1], "r+")) == NULL) {
           printf ("File cannot be opened!");
           exit (0);
       }
       fgets(str, MAX+1, fp);          /* reads a line of text */
       while (str[i] != '\0') {
           if (islower(str[i]))        /* converts into uppercase
                                       letters */
               str[i]=toupper(str[i]);
           i++;
       }
       fseek(fp, 0L, 0);              /* write it back to the
                                       same file */
       fprintf(fp, "%s", str);
       fclose(fp);
       return 0;
   }

```

Chapter 13: Storage Classes

1. The *scope* refers to the sections of code that can use the variable. In other words, the variable is visible in that section. A variable can have one of the following 3 scopes: file scope, block scope and function-prototype scope. The *linkage* determines how a variable can be used in a multiple-source file program. It identifies whether the variable can only be used in the current source file or it can be used in other source files with proper declarations. The *storage duration* determines how long a variable can exist in memory.

2. In file scope, the variable is visible from the point it is defined to the end of the file containing the definition. In block scope, the variable is visible from the point it is defined until the end of the block containing the definition. In function-prototype scope, it applies to variable names used in function prototypes. The variable is visible from the point the variable is defined to the end of prototype declaration.
3. Local variables are defined inside a function, therefore, they have function or block scope. Global variables are defined outside the functions. They have file or program scope. Functions can simplify communicate with each other using only global variables without using any function arguments. However, it is considered bad programming style if one relies on global variables. Therefore, one should avoid using global variables whenever possible.
4. The value of `q()` is 400. The value of `p()` is 30.

Chapter 14: The C Preprocessor

1. The advantages of macros are as follows. Macro arguments are not variables, they have no types nor storage allocated to them, so that one macro may serve many data types. Macros are usually faster than functions because they have no function overheads like saving variable values before function execution. The disadvantages of macros are given as follows. Macro arguments are re-evaluated at each mention in the macro body, which leads to unexpected behaviour if an argument contains side effects. Macros are substituted with the macro bodies each time they appear in the program. Macros do not check argument types. More difficult to debug programs that contain macros.
2. Inline expansion refers to text replacement at the point where a macro is called. A macro call will be replaced by the macro body defined in the macro expansion.

```
b = double(a*a);  
b = ((a*a)*2);
```

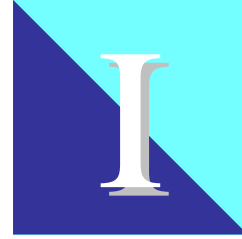
3. (a) Line 1: it means "if not defined HEADERFILE". As it is illegal to include a header file for more than one time, this is to avoid such situation happening.
Line 4: it defines a meaningful name `DEBUG` to have a value 1.
Line 6: it includes a library file "`stdio.h`". It causes the searching of the file `stdio.h` from `/usr/include`.

Line 7: it includes a file "anotherfile.h". It causes the searching of the file anotherfile.h from the user's directory, i.e. in the same directory with the file containing this header file.

(b) Lines 9 and 10 should be written as:

```
#define power(x,n,y) \  
    {int i; (y)=1; for (i=0; i < (n); i++) {(y) = (y)  
      * (x); }}
```

4. The recursive version cannot be implemented using a macro definition because the number of macro substitutions cannot be determined at compilation time.



Index

#

- #, 335–36
- ##, 336
- #define**, 48–49, 319–27
- #elif**, 331–32
- #else**, 329–30, 331–32
- #endif**, 329–30, 331–34
- #error**, 335
- #if**, 331–32
- #if defined**, 332
- #ifdef**, 329–30
- #ifndef**, 332–34
- #include**, 11, 327–29
- #line**, 335
- #pragma**, 335
- #undef**, 334

A

- abs()** function, 83
- Addition operator (+), 71
- Address operator (&), 149–50
- ALU (arithmetic logic unit), 2
- ANSI C standard, 7
- argc**, 239
- Arguments
 - in command line, 238–40
 - passing in functions, 143–45
- argv** array, 239
- Arithmetic

- assignment operators, 77–78
- assignment statement, 75–76
- operators, 71–72

Array

- as function arguments, 182–84
- declaration, 173–74
- index, 173
- initialization, 174–75
- multidimensional, 192–98
- of structures, 249–51
- relationship to pointers, 178–82

- arrow (←), 26

- ASCII (American Standard Code Information Interchange), 45

- ASCII character set, 338

- Assembler, 4

- Assembly language, 5

- atof()** function, 234

- atoi()** function, 234

- atol()** function, 234

- auto** keyword, 302

- Automatic variables, 301–2

B

- begin**, 23

Binary

- operators, 71

- Binary I/O, 286–91

- Bitwise operators, 85–86

- Braces ({}), 9

- break** statement, 102–6

- break** statement, 127–29

Bubble sort, 184–88

C

C programming language, 6–7
 Call by reference, 156–59
 Call by value, 146–49
case label, 102–6
 Case sensitive, 49
 Cast operators, 80–82
ceil() function, 83
char data type, 45
 Characters
 constants, 46
 converting, 233
 testing for, 232
 types, 45
 Closing a file stream, 279
 Command line arguments, 238–40
 Comments, 10–11
 Compiler, 5
 Compiling, 14–16
 Conditional compilation, 329–34
 Conditional operator, 106–8
const type qualifier, 47, 312–14
 Constants, 45
 character, 46
 defining, 47–49
 floating point, 46
 integer, 46
 string, 210–11
continue statement, 129–30
 Control characters, 46
 Control string
 in **printf()**, 54
 in **scanf()**, 62
 Control structures, 22–26
 repetition, 23–24
 selection, 23
 sequence, 23
cos() function, 84
 Counter-controlled loop, 110
 CPU (central processing unit), 2
ctype.h header file, 232
 CU (control unit), 2

D

Data types
 char, 45
 double, 43

float, 43
int, 42
long, 42
long double, 43
short, 42
unsigned, 42
unsigned char, 45
unsigned long, 42
unsigned short, 42

Debugging, 35–37
 Decision symbol, 22
 Declaration statements, 49
 Decomposition, 27
 Decrement operator (**--**), 72–73
default (in **switch** statement), 102–6
defined, 332
 Division operator (**/**), 71
 Documentation, 38–39
double data type, 43
do-while statement, 124–27
 flowchart, 124

E

Editor, 4
end, 23
enum keyword, 266
 Enumerated data type, 266–68
EOF (end-of-file), 62, 276
 Errors
 logic, 35
 run-time, 35
 syntax, 34
 Escape sequence, 46
exp() function, 84
extern keyword, 303
 External variables, 303–4

F

fabs() function, 83
fclose() function, 279
feof() function, 297
fflush() function, 66, 274
fgetc() function, 281
fgets() function, 283
FILE pointer, 277
 Files
 access modes, 277
 binary mode, 275

- closing, 279
- copying, 282
- Formatted I/O, 285–86
- offset, 292
- opening, 277–78
- position marker, 291
- random access, 291–96
- reading, 278–79
- text mode, 275
- writing, 278–79
- float** data type, 43
- float.h* header file, 44
- Floating point
 - constants, 46
 - types, 43–45
- floor()** function, 83
- Flowcharts, 21–22
- Flowlines, 22
- fopen()** function, 277
- for** statement, 118–24
 - flowchart, 119
- Formatted input, 61–66
- Formatted output, 54–61
- fprintf()** function, 278, 285
- fputc()** function, 282
- fputs()** function, 284
- fread()** function, 287
- fscanf()** function, 278, 285
- fseek()** function, 292
- ftell()** function, 293
- Functions
 - arguments, 143–45
 - body, 138
 - call by reference, 156–59
 - call by value, 146–49
 - calling, 143–45
 - definition, 136–41
 - formal arguments, 137
 - header, 137
 - parameters, 137
 - prototypes, 142–43
 - recursive, 159–65
 - returning structures, 260
 - structures as arguments, 255–59
- fwrite()** function, 288

G

- getc()** function, 281
- getchar()** function, 281
- gets()** function, 218–19, 284
- Global variables, 306–8

H

- Hardware, 1–3
- High-level language, 5

I

- if** statement, 93–95
 - flowchart, 94
 - nested-**if**, 99–102
- if-else** statement, 95–96
 - flowchart, 96
- Increment operator (**++**), 72–73
- Indirection operator (*****), 151–55
- Infinite loop, 120
- Input/output devices, 3
- Input/output symbol, 22
- int** data type, 42
- Integers
 - constants, 46
 - types, 42–43
- isalnum()** function, 232
- isalpha()** function, 232
- isctr1()** function, 232
- isdigit()** function, 232
- isgraph()** function, 232
- islower()** function, 232
- isprint()** function, 232
- ispunct()** function, 232
- isspace()** function, 232
- isupper()** function, 232
- isxdigit()** function, 232

K

- Keywords, 49

L

- limits.h* header file, 43
- Linker, 15
- Linking, 14–16
- Local variables, 306–8
- log()** function, 84
- log10()** function, 84
- Logic errors, 35
- Logical expressions, 92
- Logical operators, 90–93
 - Logical **and** (**&&**), 91
 - Logical **not** (**!**), 91

- Logical **or** (**|**), 91
 - precedence of, 92
- long** data type, 42
- long double** data type, 43
- Loops
 - counter-controlled, 110
 - do-while**, 124–27
 - for**, 118–24
 - nested, 131–33
 - sentinel-controlled, 111
 - while**, 111–18
- Lvalue, 76

M

- Machine language, 5
- Macros, 319–27
 - definition, 319
 - expansion, 319
 - with arguments, 322–27
- Main memory, 2
- main()** function, 8
 - arguments in, 238–40
- math.h* header file, 83
- Mathematical functions, 83–84
- Matrix, 192
- Member operator (**.**), 248
- Members of structures, 245
- Modular design, 166
- Modulus operator (**%**), 71
- Multidimensional arrays, 192–98
 - as function arguments, 198–207
 - relationship to pointers, 197–98
- Multiplication operator (*****), 71

N

- Negative operator (**-**), 71
- Nested-**if** statement, 99–102
- Newline character, 46
- NULL**, 151
- Null character, 210

O

- Object code, 14
- Opening a file stream, 277–78
- Operators, 71
 - arithmetic, 71–72
 - arithmetic assignment, 77–78
 - binary, 71

- bitwise, 85–86
- decrement, 72–73
- increment, 72–73
- logical, 90–93
 - precedence of, 74
- relational, 90–93
- shift, 86
- ternary, 71
- types of, 71
- unary, 71

P

- Parameter list, 137
- Parentheses in arithmetic operators, 74
- Pointer, 149–56
 - declaration, 150
 - initialization, 151
 - passing to function, 156–59
 - relationship to arrays, 178–82
 - to **FILE**, 277
 - to structure, 253–55
 - variables, 150–51
- Positive operator (**+**), 71
- Postfix operators, 72
- pow()** function, 83
- Precedence of operators, 74, 92
- Prefix operators, 72
- Preprocessor, 316–19
- Preprocessor directives
 - #define**, 319–27
 - #elif**, 331–32
 - #else**, 329–30, 331–32
 - #endif**, 329–30
 - #error**, 335
 - #if**, 331–32
 - #if defined**, 332
 - #ifdef**, 329–30
 - #ifndef**, 332–34
 - #include**, 327–29
 - #line**, 335
 - #pragma**, 335
 - #undef**, 334
- Preprocessor macros, 319–27
- Primary storage, 3
- printf()** function, 54–61, 222–23
 - argument list, 55
 - control string, 54
 - conversion specification, 56–57
 - conversion specifier, 56
 - field width, 57
 - flags, 56

- precision field, 57
- size specification, 57
- Process symbol, 21
- Program
 - debugging, 35–37
 - design, 20
 - development process, 18–20
 - documentation, 38–39
 - maintenance, 20
 - modification, 20
 - testing, 37–38
- Programming Language, 5–6
- Pseudocode, 22
- putc()** function, 282
- putchar()** function, 282
- puts()** function, 222, 284

R

- RAM (random access memory), 3
- Random access files, 291–96
- register** keyword, 305
- Relational expressions, 92
- Relational operators, 90–93
 - precedence of, 92
- Repetition, 23–24
- return** statement, 139–41
- rewind()** function, 293
- ROM (read only memory), 3
- Run-time errors, 35
- Rvalue, 76

S

- scanf()** function, 61–66, 219–21
 - address list, 62
 - argument list, 62
 - control string, 62
 - conversion specification, 62
 - conversion specifier, 62
 - field width, 63
 - flags, 62
 - size specification, 63
- Secondary storage, 3
- Sentinel-controlled loop, 111
- Shift operators, 86
- short** data type, 42
- sin()** function, 84
- sizeof** operator, 173, 224
- Software, 3–5
- Source code, 6, 13

- sprintf()** function, 235
- sqrt()** function, 83
- sscanf()** function, 235
- Standard
 - error stream, 271
 - input stream, 271
 - output stream, 271
- static** keyword, 304
- Static variables, 304–5
- stderr**, 271
- stdin**, 271
- stdio.h* header file, 53
- stdlib.h* header file, 234
- stdout**, 271
- Stepwise refinement, 27
- strcat()** function, 225–26
- strchr()** function, 231
- strcmp()** function, 226–29, 226–29
- strcpy()** function, 229–31
- Streams, 270–75
 - buffering, 273–75
 - I/O, 271
 - redirection, 272–73
 - standard, 271–72
- string.h* header file, 223
- Strings
 - array notation, 211–12
 - array of, 237
 - array of pointers, 236
 - comparing, 226–29
 - concatenating, 225–26
 - converting to numbers, 234–35
 - copying, 229–31
 - declaring, 211–18
 - formatted input and output, 235–37
 - initializing, 211–18
 - input, 218–21
 - length of, 224–25
 - library functions for, 223–24
 - output, 222–23
 - pointer notation, 213–14
 - searching, 231–32
 - variables, 211–18
- strlen()** function, 224–25
- strncat()** function, 225–26
- strncmp()** function, 226–29
- strncpy()** function, 229–31
- strrchr()** function, 231
- strstr()** function, 231
- struct** keyword, 245
- Structures
 - accessing members, 248

- as function arguments, 255–59
- assignment, 249
- creating, 245–46
- declaring, 246
- in functions, 255–60
- initializing, 248
- member operator (`.`), 248
- nested, 251–53
- pointer operator (`->`), 254
- pointers to, 253–55
- tag, 247
- template, 245–46
- variable, 247

Structures

- array of, 249–51
- as return values, 260

Subscripts, 173

Subtraction operator (`-`), 71

switch statement, 102–6

Syntax errors, 34

T

Tag (of a structure), 247

tan() function, 84

Template (of a structure), 245–46

Terminal symbol, 22

Terminating null character, 210

Ternary operators, 71

Text editor, 4

tolower() function, 233

Top-down stepwise refinement, 27–31

toupper() function, 233

Two-dimensional arrays, 192

Type casting, 80–82

typedef statement, 260–62

U

Unary operators, 71

ungetc() function, 296

union keyword, 262

Unions, 262–66

unsigned char data type, 45

unsigned data type, 42

unsigned long data type, 42

unsigned short data type, 42

V

Variables, 49–51

- automatic, 301–2
- external, 303–4
- global, 306–8
- linkage, 300–301
- local, 306–8
- naming, 49
- register, 305–6
- scope of, 300
- static, 304–5
- storage duration, 301

void

- as return type, 137
- in parameter list, 137

volatile type qualifier, 312

W

while statement, 111–18

- flowchart, 112

Whitespace, 62