ELEC S212

Network Programming and Design

Unit 5

Introduction to network application development

香港公開大學
THE OPEN UNIVERSITY
OF HONG KONG

科技學院 *School of Science and Technology*

Course team

Developers:	Jacky Mak, Consultant
Designer:	Ross Vermeer, ETPU
Coordinator:	Dr Philip Tsang, OUHK
Member:	Dr Steven Choy, OUHK

External Course Assessor

Prof. Cheung Kwok-wai, The Chinese University of Hong Kong

Production

ETPU Publishing Team

The Open University of Hong Kong
Ho Man Tin, Kowloon
Hong Kong

This course material is printed on environmentally friendly paper.

Contents

Overview

This unit introduces you to C programming in the Linux environment. The C programming skills and knowledge that you will acquire in this unit build a foundation for the network programming that you will do in the next two units.

This unit consists of eight main sections. The first section introduces to you basic concepts in C programming. We briefly discuss the history of C, and the virtues that have helped to make it a popular and important programming language. Since you may have no programming experience, this section also briefs you on the basic programming skills needed. We also describe the basic steps required to create C programs in the Linux environment, and analyse a few simple programs in details to help you get familiar with the basics of C programming.

The second section presents you with basic elements of the C language, including variables, constants, declarations, statements and expressions.

The third section discusses program sequences, and control of the C language. The control-flow statements of a language specify the order in which computations are preformed.

The fourth section teaches the need for writing your own functions, which are modules of code that you execute and control from the `main` function. This section stresses the use of structured programming.

The fifth section presents you with the concepts of arrays and pointers of the C language. An array is a collection of similar data in an ordered sequence. A pointer is a variable that contains the address of a variable. Pointers and arrays are closely related, so this section also explores this relationship.

The sixth section discusses the concept of structures in C language. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures help organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

The seventh section describes the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs. This section concentrates on input and output.

The final section illustrates some common mistakes often made by beginners in C programming.

In short, this unit:

- describes the basic steps in creating C programs;
- describes different data structures in C;

- describes variables, types and type declarations in C;
- discusses the basic input and output functionalities in C;
- discusses the basic conditional-testing statement and looping mechanism in C;
- describes the arithmetic operations in C;
- explains the concepts of using arrays and pointers in C; and
- explains the application development and develops applications using C.

Programming examples are included throughout the unit. You will find that reading the examples helps you become familiar with the language very quickly. There are also activities in each section to give you more practice.

The best way to learn programming is to write real programs. As practice is more beneficial than reading guidelines, you are strongly recommended to enter the programs in this unit, compile them, execute them, and modify them. There is indeed no better way to get familiar with a programming language than to write a lot of programs in it!

This unit is intended to take you five weeks (or approximately 40 hours) to complete.

Introduction to application programming using C

The C programming language is one of the most important and popular programming languages in the history of computing. It was developed in the years 1969–1973 by Dennis Ritchie at Bell Telephone Laboratories.[1] Although C was originally designed for implementing system software (the UNIX operating system, in particular), it has also been widely used for developing portable application software. Since its appearance in the early 1970s, C has won widespread acceptance because it possesses the following virtues:

- *Modern design features* — C incorporates control features that are desirable in the theory and practice of computer science. That is, C encourages top-down planning, structured programming and modular design, resulting in more reliable and understandable programs.

- *Efficiency* — C is an efficient language that takes advantage of the capabilities of modern computers. Programs written in C can be fine-tuned for maximum speed or most efficient use of memory. As a result, C programs tend to be compact and to run quickly.

- *Portability* — Compared to other programming languages, C is probably the most portable. Compilers for C exist on UNIX, GNU/Linux, BSDs, Microsoft Windows, Mac OS X, embedded systems, and many other platforms that you might have never heard of. C Programs written on one platform can often be recompiled and executed on other platforms with little or no modification. If modifications are necessary, they can often be made by simply changing a few minor spots in the header files accompanying the main program.

- *Power and flexibility* — C is powerful and flexible. In fact, many operating systems, including UNIX and Microsoft Windows, are written in C. C is often used to implement compilers and interpreters for other programming languages, such as FORTRAN,[2] Perl,[3]

[1] http://www.alcatel-lucent.com/wps/portal/BellLabs

[2] FORTRAN (FORmula TRANslation) is a general-purpose programming language introduced in 1957 that is especially suited to numeric computation and scientific computing. It is still in active use in computationally intensive areas and high-performance computing.

[3] Perl is a high-level, general-purpose, interpreted, dynamic scripting language designed by Larry Wall in 1987. Perl borrows features from other programming languages including C, shell scripting, AWK, and sed. The language provides powerful text processing facilities without the arbitrary data length limits of many contemporary UNIX tools, facilitating easy manipulation of text files.

Python[4] and BASIC.[5] C programs are also widely used for solving physics, engineering and mathematics problems. In addition, many industrial control programs and embedded systems are also implemented with C.

- *Programmer oriented* — C was designed to fulfill to the specific needs of the programmers of the UNIX operating system. It gives programmers almost unlimited access to hardware, and it allows programmers to manipulate individual bits in memory. It has a highly expressive syntax and a rich selection of operators and that allow programmers to write highly succinct and efficient programs in C. As a result, many tasks, such as converting forms of data, can be accomplished much more easily in C.

A brief history of C

In *Unit 4* you were provided with a brief history of the UNIX operating system. You learned that UNIX was first developed in 1969 by Ken Thompson, Dennis Ritchie and their colleagues at Bell Labs.

The origin of C is closely tied to the early development of UNIX. UNIX was originally written in assembly language on a PDP-7, an 18-bit minicomputer produced by Digital Equipment Corporation (DEC) in the late 1960s. The applications that ran on the original UNIX were written in a mix of assembly language and an **interpreted language** called B, which was basically a stripped-down version of another language called BCPL (Basic Combined Programming Language).[6]

Traditional C

In 1970, the UNIX group at Bell Labs acquired a new, more powerful PDP-11 computer. Ken Thompson and Dennis Ritchie decided to port UNIX to the new computer, and he needed to rewrite the operating system in a high-level language. However, other high-level languages at

[4] Python is another high-level, general-purpose, interpreted, dynamic scripting language designed by Guido van Rossum in 1991. It is becoming more and more popular in the last few years.

[5] BASIC (Beginner’s All-purpose Symbolic Instruction Code) is a family of high-level programming languages that first appeared in the mid-1960s. It was originally intended as a programming language for non-science students, but it has become extremely popular on microcomputers in the last few decades.

[6] BCPL was created by Martin Richards of the University of Cambridge in 1966. It was originally intended for compiler writing and systems programming.

the time (FORTRAN, COBOL,[7] ALGOL,[8] PL/I,[9] etc.) were too slow to use for system programming. On the other hand, B also lacked certain functionalities needed to take advantage of PDP-11's new architecture and powerful features. Therefore, Dennis Ritchie started to add features and data types to B, and ended up with an early version of a new language that he named C. By 1973, with the addition of `struct` types (which you will learn about later in this unit), the C language had become powerful enough that most of the UNIX kernel was rewritten in C. Today, we often refer to this early version of C as traditional C.

K&R C

In 1978, Dennis Ritchie and Brian Kernighan published the first edition of *The C Programming Language*. This book, commonly known as K&R, served for many years as an informal specification of the C programming language. Today, we commonly refer to the version of C described in K&R as K&R C.

ANSI C (C89) and ANSI/ISO C (C90)

During the late 1970s and the 1980s, with the release of many C compilers for a wide variety of mainframes, minicomputers and microcomputers (including the IBM PC) and the increasing popularity of UNIX, C's popularity increased significantly. The rapid growth of C led to the development of different versions of the language that were largely similar, but often incompatible. In 1983, the American National Standards Institute (ANSI) formed a committee to define a standard for C. The committee approved a version of C in 1989 which is now known as ANSI C, or simply C89. In the following year, the ANSI C standard was adopted by the International Organization for Standardization (ISO). This version of C, called ANSI/ISO C, is basically the same as ANSI C. It is sometimes referred to as C90.

C89 is supported by all current C compilers, and most C code being written nowadays is based on it. Any program written in ANSI C and without any hardware-dependent code will run correctly on any platform with a conforming C implementation.

[7] COBOL (COmmon Business-Oriented Language) is one of the oldest programming languages that is still in active use today. It was created in 1959 and was designed primary for business, finance and administrative systems for large organizations.

[8] ALGOL (ALGOrithmic Language) is a family of imperative programming language introduced in 1958 which greatly influenced many other languages (including BCPL, B, Pascal, C and Simula). It became the de facto way algorithms were described in textbooks and academic works.

[9] PL/I (Programming Language One) is an imperative programming language introduced in 1964 that was designed for scientific, engineering and business applications.

C99

Since the first standardization of the C programming language, additional changes have been added to it. The most recent standard of C was adopted in 1999, and is known as ANSI C99 or simply C99.

The following figure illustrates the history of the C language:

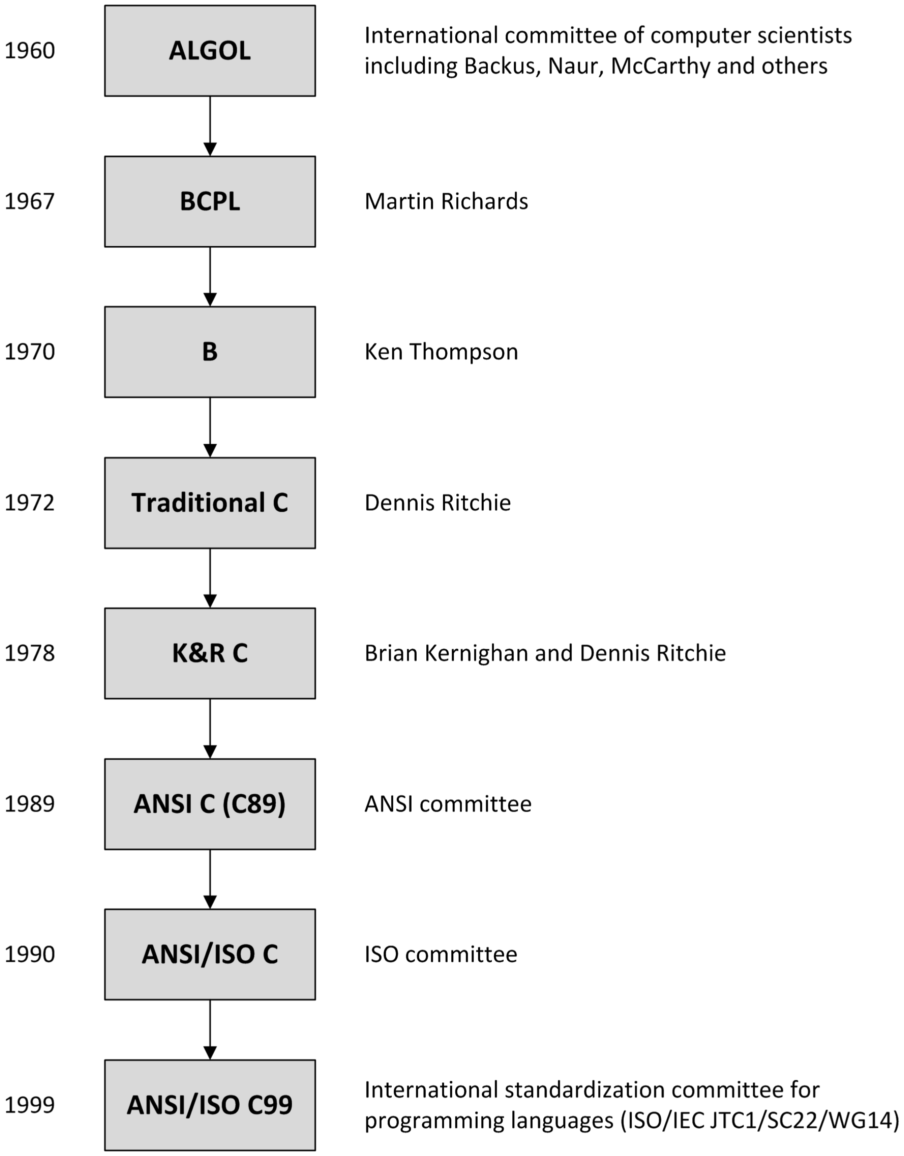

1960
ALGOL
International committee of computer scientists including Backus, Naur, McCarthy and others
1967
BCPL
Martin Richards
1970
B
Ken Thompson
1972
Traditional C
Dennis Ritchie
1978
K&R C
Brian Kernighan and Dennis Ritchie
1989
ANSI C (C89)
ANSI committee
1990
ANSI/ISO C
ISO committee
1999
ANSI/ISO C99
International standardization committee for programming languages (ISO/IEC JTC1/SC22/WG14)

Figure 5.1 History of the C language

The following reading is a short paper written by Dennis Ritchie on the early development of the C language. Although this reading is optional, you are highly recommended to read this fascinating story of one of the most important and popular programming languages in the history of computing.

Reading 5.1 (online)(optional)

Ritchie, D (1993) *The Development of the C Language*, Chistory:

https://www.bell-labs.com/usr/dmr/www/chist.html

Programming basics

You may find programming basics a bit annoying if this is your first go at writing a program. The function of programming is to tell the computer what to do, and how to do it. A computer is really not very 'clever.' You must give it step-by-step instructions to get it to do something for you. It may fail to achieve the task if it misses even a single step. The following example should give you an idea of how a computer works.

Example 5.1

Let's say you want a computer to find the largest number among the following numbers:

10, 15, 100, 20

If you ask a primary school student to do this task, he or she might just pick '100' out without thinking. However, the computer has to do it in a different way. Unlike a human being, the computer cannot browse all of the numbers at the same time. It reads each number sequentially. So, the steps for a computer to do this task may be as follows:

1. Read the first number and store it in its memory location A.

2. Read the second number and compare it with the number in memory location A.

3. If the second number is larger than the number in memory location A, replace the number in memory location A with the second number.

4. Read the third number and compare it with the number in memory location A.

5. If the third number is larger than the number in memory location A, replace the number in memory location A with the third number.

6. Read the fourth number and compare it with the number in memory location A.

7 If the fourth number is larger than the number in memory location A, replace the number in memory location A with the fourth number.

8 Print the number in memory location A (which will be the largest number).

How does this make you feel? It's true — you really do have to tell the computer all of these steps just for such a simple task! You may feel that the steps shown in the above example are quite artificial, but that really is the way a computer 'thinks'! So, to learn programming, you have to learn to think in the same way as a computer.

The steps in the example are written in English. In real programming, however, we need to write in the syntax of a programming language.

In the following reading, Steve Summit presents a very good introduction to computer programming for newcomers.

Reading 5.2 (online)

Summit, S (1995–96) 'C programming: a short introduction to programming':

http://www.eskimo.com/~scs/cclass/progintro/top.html

You can focus on the following sections in the reading:

- Skills needed in programming: This link introduces the most important skills in programming.

  http://www.eskimo.com/~scs/cclass/progintro/sx1.html

- Real programming model: This link explains the concepts of code and data in a computer program.

  http://www.eskimo.com/~scs/cclass/progintro/sx3.html

- Elements of real programming languages: This link describes the elements that programming languages typically contain (e.g. variables, expressions, assignments, conditional, types, statements, control flow constructs, functions, etc.). These elements are found in all languages, not just C.

  http://www.eskimo.com/~scs/cclass/progintro/sx4.html

Self-test 5.1

1 In the above reading, four skills are needed in programming. Name them.

2 A computer program consists of two parts: *code* and *data*. Write your own definitions of *code* and *data*.

Basic steps in creating a C program

C is a **compiled language**. This means that a special program, called the **compiler**, is required to analyse C **source code** and translate it to machine code that can be directly executed by a computer. This analysis and translation process is known as compilation. Compilation is required only once for each platform that the program is targeted to run. Once the executable program is produced by the compiler, it can be executed on the targeted platform as many as times as desired.

The following figure shows the steps that are involved in entering, compiling, and executing a C program, and the typical UNIX commands that would be entered from the command line:

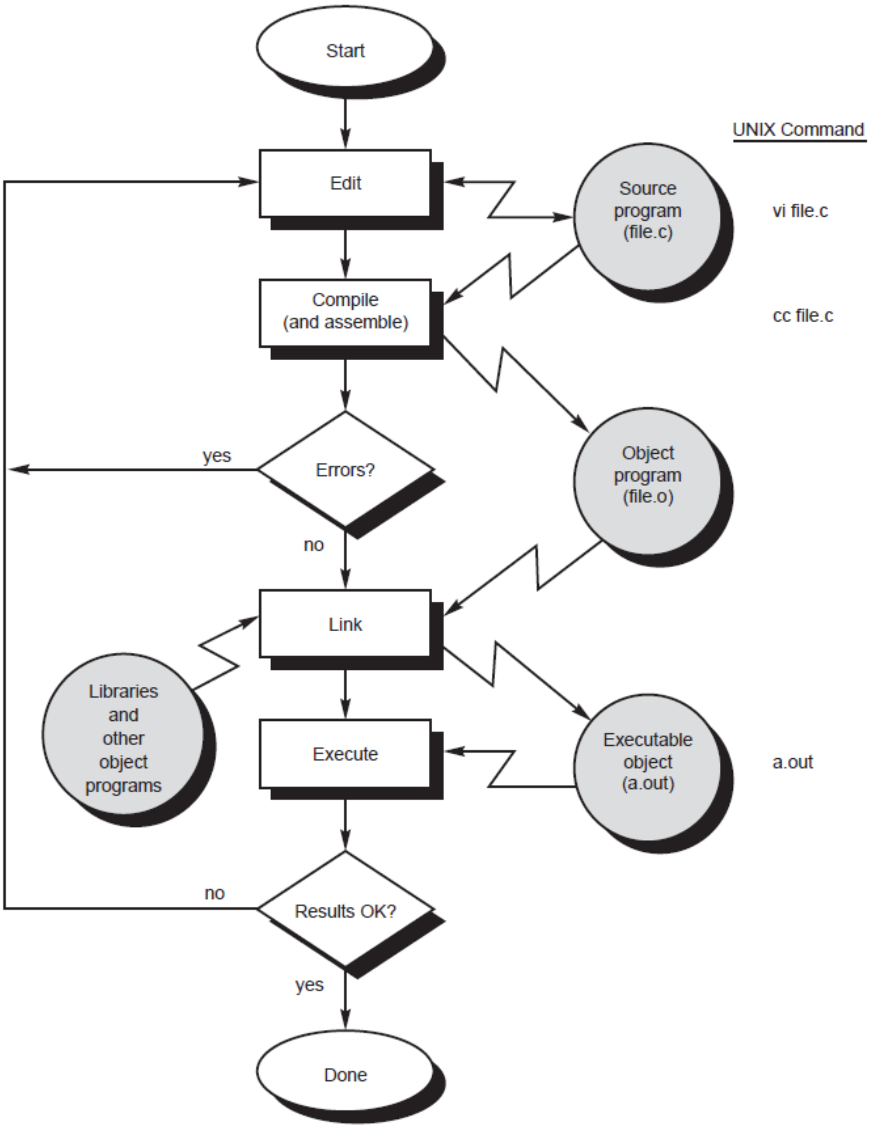


Figure 5.2 Typical steps for entering, compiling, and executing C programs from the UNIX command line (Kochan 2005, Figure 2.1)

The C source code is first typed into a file, which is called the **source file**. By convention, programmers name C source files with the `c` extension, although this is not really required by most implementations. The source file contains only plain text that can be edited using any plain text editor. On a UNIX or Linux system, you typically use a text editor such as `vi` or `pico` to enter the program.

Alternatively, you can use an **integrated development environment** (**IDE**) to enter and edit C programs. An IDE is a software application that provides comprehensive facilities to programmers for software development. It normally consists of a source code editor, a **compiler** and/or an **interpreter**, build automation tools, and a debugger. Using an IDE can greatly increase a programmer's productivity.

After the source code is entered, it can be compiled. On a UNIX system, the command to initiate the compilation process is called `cc`. On a Linux system, you use the popular GNU C compiler[10]; the command is `gcc`.

The first phase of the compilation process is *preprocessing*. In the preprocessing phase, the compiler performs the following tasks as specified in the C standard:

- *Trigraph replacement* — the preprocessor replaces trigraph sequences with the characters they represent.

- *Line splicing* — physical source lines that are continued with escaped newline sequences are *spliced* to form logical lines.

- *Tokenization* — the preprocessor breaks the result into *preprocessing tokens* and whitespace. It replaces comments with whitespace.

- *Macro expansion* and *directive handling* — these tasks handle *preprocessor directives* such as source file inclusion (`#include`), macro definitions (`#define`) and conditional inclusion (`#if`).

Depending on the particular C implementation on the system, preprocessing can be performed by the compiler itself, or by a separate **C preprocessor** program invoked by the compiler.

After the preprocessing phase, the compiler begins to inspect each program statement contained in the source code to check whether it conforms to the syntax and semantics of the C language. If any errors are discovered during this phase, they are reported to the user, and the compilation process is aborted immediately. If no syntactic and semantic errors are found, then the compiler proceeds to translate the source code into an equivalent **assembly language** program targeted to the host system.

After the program has been translated into an equivalent assembly language program, the next step is to translate the assembly language statements into actual machine code. Again, this step might or might not involve the execution of a separate program known as an **assembler**. On most systems, the assembler is executed automatically as part of the compilation process. The output of this phase is one or more object files that contain the equivalent machine code of the program. Object files are typically named with the `o` extension.

The final phase of the compilation process is linking. The purpose of the linking phase is to get the program into a final form for execution on the

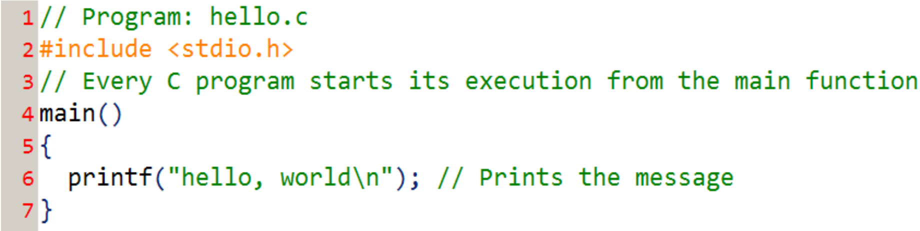
[10] The **GNU Compiler Collection** (GCC) (http://gcc.gnu.org) is a compiler system produced by the GNU Project (http://www.gnu.org). Originally named the GNU C Compiler, because it only handled the C programming language, GCC 1.0 was released in 1987. It was later extended to support C++, FORTRAN, Pascal, Java and other programming languages. Nowadays, GCC is the standard compiler on most modern UNIX-like operating systems, including GNU/Linux, BSDs and Mac OS X.

computer. If the program consists of multiple source files, the corresponding object files, together with the appropriate object files from the system's code library, are linked together by a **linker** to produce the final **executable file**, which can run by a user directly from the command line. Under UNIX, the executable program is named `a.out` by default, although you can specify its name when you invoke the compiler. Under Microsoft Windows, an executable file usually has the same name as the source file, but with the `c` extension replaced by the `exe` extension.

In reality, most implementations do not require the programmer to carry out the above compilation steps manually. Instead, the compiler can often produce the executable file directly by simply specifying the source files and any necessary precompiled object files.

Activity 5.1

The first edition of the Kernighan and Ritchie book uses the 'hello, world' program as an introductory program. The program (with added comments) is shown in the following listing:



```
// Program: hello.c
#include <stdio.h>
// Every C program starts its execution from the main function
main()
{
  printf("hello, world\n"); // Prints the message
}
```

Listing 5.1 The 'hello, world' program

Don't worry if you do not understand this program now. We will analyse this program in details later in this unit.

To compile the 'hello, world' program using the command line, follow these steps:

1 If you have completed Activity 4.1, launch your Ubuntu desktop. Open a command line window by clicking Applications > Accessories > Terminal. Then perform the following steps inside this command line window.

If you have not completed Activity 4.1, or if you prefer, you may connect to the course Linux server as described in Lab 1.1 of the 'Lab Book' (Kwan et al. 2009) to perform the following steps.

2 Use a text editor, such as `vi`, to enter the program as shown in Listing 5.1. Save the source file as `hello.c`.

Note that the numbers shown in the first column indicate the line numbers, which are not part of the program source code. The line numbers are included in the listing for the sake of our later discussion only. Do not enter the line numbers in your source file. Also note that C is a case-sensitive language, so make sure you enter the program character-by-character in the exact case, as shown in Listing 5.1.

3 Execute the command `gcc -E -o hello.i hello.c` to preprocess `hello.c` and then stop. The preprocessing output is stored in a file named `hello.i`. Use a text editor to inspect the contents of `hello.i`. You should notice that the preprocessing output file `hello.i` contains many more lines than the original source file `hello.c`. Can you locate the original source code that you enter in `hello.c`? Can you find the comments that you entered in lines 1, 2, 3, 6 and 9 in the original source file?

4 Execute the command `gcc -S hello.i` to translate the source code into the equivalent assembly language program, using the preprocessing output as the input to `gcc`. Use a text editor to inspect the output file `hello.s`. The content of this file is the equivalent assembly language code for the original program. Does the content make any sense to you?

5 Execute the command `gcc -c hello.s` to translate the assembly language program into **object code**. Use a text editor to inspect the object file `hello.o`. Does the content of this file make any sense to you?

6 Execute the command `gcc hello.o` to produce the final executable file. Since no output file is specified, the resulting executable file will be named `a.out`.

7 Run the executable file by entering `./a.out`. What is the output of the program?

8 You can also produce the executable file in one single command. Execute the command `gcc -o hello hello.c` to produce the executable file named `hello`. Execute the program by entering `./hello`. Does it produce the same output as `a.out`?

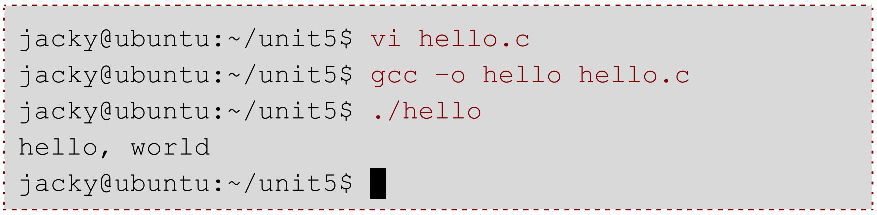
Dissecting simple C programs

In this subsection, we introduce the basic concepts of C programming by analysing a few simple C programs in details. The first one we discuss is the ‘hello, world’ program shown in Listing 5.1. The program is listed here again:

```
// Program: hello.c
#include <stdio.h>
// Every C program starts its execution from the main function
main()
{
  printf("hello, world\n"); // Prints the message
}
```

Listing 5.2 The ‘hello, world’ program

The following figure shows the output of the ‘hello, world’ program:



```
jacky@ubuntu:~/unit5$ vi hello.c
jacky@ubuntu:~/unit5$ gcc -o hello hello.c
jacky@ubuntu:~/unit5$ ./hello
hello, world
jacky@ubuntu:~/unit5$ █
```

Figure 5.3 Editing, compiling, and executing the ‘hello, world’ program.

The ‘hello, world’ program, though simple, shows the basic structure of a C program.

The main function

All C programs must contain one and only one function with the name: `main`. Execution of all C programs start from the first statement of the `main` function, and continues until the last one.

You can therefore consider a function to be the grouping of a set of statements to form a single unit.[11] The statements of a function are put inside the braces `{}` and separated by the semi-colon `;`. The function name must be followed with brackets `()`. Arguments, if any, for a function should be inside these brackets. Note also that a function may need input values for processing. These input values may be passed to the function through its arguments. If there is no argument for the function, it is left blank inside the brackets, just like `main()`. Note that sometimes the term *parameter* may be used instead of *argument*.

The `printf` function and escape sequences

In addition to the required `main` function, other functions are almost always used in a typical C program. For example, in the ‘hello, world’ program, `printf`, which is a function for printing the results on the screen, is used. In line 9,

```
printf(“hello, world\n”);
```

is a statement calling the function: `printf`. The first argument of `printf` is the string enclosed in the double quotes. We use ‘the first argument’, but not ‘the only argument’ to describe the string, because later you will find that there can be more than one argument for `printf`. `“hello, world\n”` is the input value passed to the function through the argument. `printf` will print the string out after formatting.

[11] In computer science, a function ‘is an abstract entity that associates an input to a corresponding output according to some rule.… It is a portion of code within a larger program, which performs a specific task and can be relatively independent of the remaining code’ (Source: Wikipedia — Function (Computer Science)).

You may have noticed the special sequence `\n` at the end of the string. This special sequence represents the newline character. That means it will force the output after that onto a new line. That is why the statement `printf(“hello, world\n”);` prints the string: `hello, world` on a new line. You can delete the `\n` characters from the first `printf` statement to see what the output will become.

The backslash `\` has special meaning in strings. It is called the escape character and, together with the character following it, it is called an escape sequence. `\n` is one of these escape sequences. `\` causes the character following it to be interpreted differently by the compiler. In practice, then, `\n` is treated as a single character. Note that not all characters can form an escape sequence with the backslash. Some of these combinations are introduced here.

Suppose you want to print the following string on the screen using `printf`:

```
The sequence \n means the “newline character”.
```

What will be the corresponding `printf` statement? Is the following statement correct?

```
printf(”The sequence \n means the “newline character”. ”);
```

Obviously, this statement is incorrect, since the double quotes are not matched and the `\n` will force a newline, instead of printing out `\n`. Here, the escape character has to be used for the quotation mark “ and the backslash `\`. The correct `printf` statement should be:

```
printf(”The sequence \\n means the \“newline character\”. ”);
```

`\\` and `\”` are the escape sequences for `\` and “ respectively. `\t` is another commonly-used escape sequence; it causes a tab to be printed.

Preprocessor directives

You may also have noticed the preprocessor directive `#include <stdio.h>` in line 4 of the program. This preprocessor directive tells the compiler to include the information in the standard input/output library. Every function used in the program should be defined beforehand. `printf` is no exception. `printf` is actually a library function provided by C, and its information can be found in the header file `stdio.h` . That is why you can use `printf` without defining it. Details of the function definition in C are discussed later in this unit.

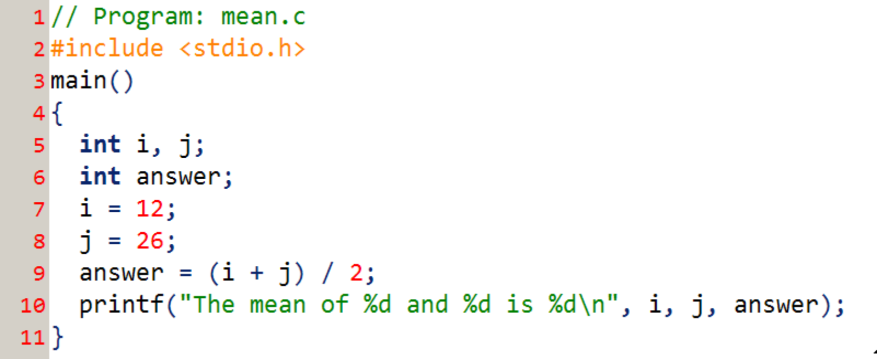
Comments

Sometimes it is difficult to trace what the logic of a program is from the source code. This is usually true for large programs, and for programs that were not written by you originally. Comments help a human reader (including the programmer who wrote the code originally) to understand the code more easily. In C, there are two forms of comments:

- `/* … */` — as shown on lines 1–3 in Listing 5.2. Anything between `/*` and `*/` comprises comments that are ignored by the compiler.

- `//` — as shown on lines 6 and 9 in Listing 5.2. Anything starting from `//` until the end of the current line is ignored by the compiler.

Now consider another simple program ‘mean’, as shown in the following listing:



```
 1 // Program: mean.c
 2 #include <stdio.h>
 3 main()
 4 {
 5   int i, j;
 6   int answer;
 7   i = 12;
 8   j = 26;
 9   answer = (i + j) / 2;
10   printf("The mean of %d and %d is %d\n", i, j, answer);
11 }
```

Listing 5.3 The ‘mean’ program

The ‘mean’ program calculates the mean of two numbers and displays the mean on the screen:

```
jacky@ubuntu:~/unit5$ vi mean.c
jacky@ubuntu:~/unit5$ gcc -o mean mean.c
jacky@ubuntu:~/unit5$ ./mean
The mean of 12 and 26 is 19
jacky@ubuntu:~/unit5$ █
```

Figure 5.4 Editing, compiling, and executing the ‘mean’ program.

Variables, types and declarations

In most programs, you need to use variables to store the values used during the program’s execution. In C, variables must be declared before use. The data type of a variable has to be defined in the declaration. The data type of a variable limits the kinds of value the variable can take on. The following two statements in lines 5 and 6 of ‘mean’ are variable declarations:

```
int i, j;
int anwser;
```

`i`, `j` and `answer` are variable names. `int` defines the variables as integer type, which means they can only take on integer values. Details of the data type are discussed later in this unit. The declaration of the same data type can be done on the same line, so the above two statements may be combined to:

```
int i, j, answer;
```

Statements

The next two lines are assignment statements:

```
i = 12;
j = 26;
```

`i` and `j` will take on the values 12 and 26, respectively, after executing these two statements.

The following statement is also an assignment statement:

```
answer = (i + j) / 2;
```

The result of the mathematical expression `(i + j) / 2` will be assigned to the variable `answer`. In C, the mathematical operators `+`, `-`, `*` and `/` are used, and are described in more detail later in the unit.

Formatted output

There is also a `printf` statement in this example. But this time, a different format is used. You may find there are four arguments for `printf` this time, each separated by a comma:

```
printf("The mean of %d and %d is %d\n", i, j, answer);
```

The first argument is still a string embedded by the double quotes. The remaining three arguments are integer variables. Inside the string, you may find the special sequence `%d`. Whenever `printf` processes the `%` sign, `printf` will not print the exact string out. Instead, `printf` will read the next arguments in order to decide what to print. The character following the `%` sign also tells `printf` what type of argument should be printed out. It is `%d` in this example.

For `%d`, `printf` will know that a variable of integer type should be supplied to it, and it will print out a decimal number. `printf` will read the second argument, which is `i`, and substitute it for the first occurrence of `%d` in the string. In the same way, `j` will be substituted for the second occurrence of `%d` and `answer` for the third occurrence. That is why the output is:

```
The mean of 12 and 26 is 19
```

instead of:

```
The mean of %d and %d is %d
```

Activity 5.2

Modify `mean.c` to calculate the average of the following three numbers:

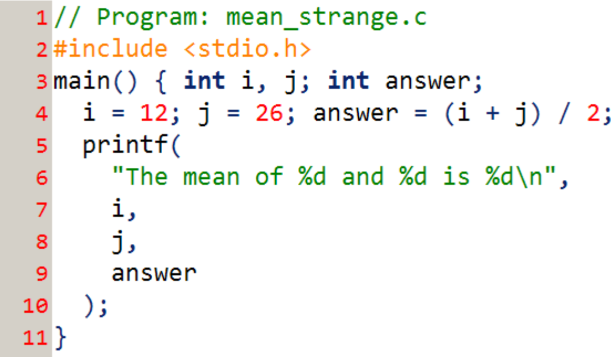
```
12345, 23456, 34567
```

Name the new program `mean3.c`. The program should print the following sentence when executed:

```
The average of the 3 numbers: 12345, 23456 and 34567 is 23456.
```

Free format

C is a free format language. This means the compiler does not care about the arrangement of the code. You can break a line anywhere except in the middle of a variable name and string, add any number of spaces between expressions, and indent lines as you like. For example, the ‘mean’ program can be written in the following way:

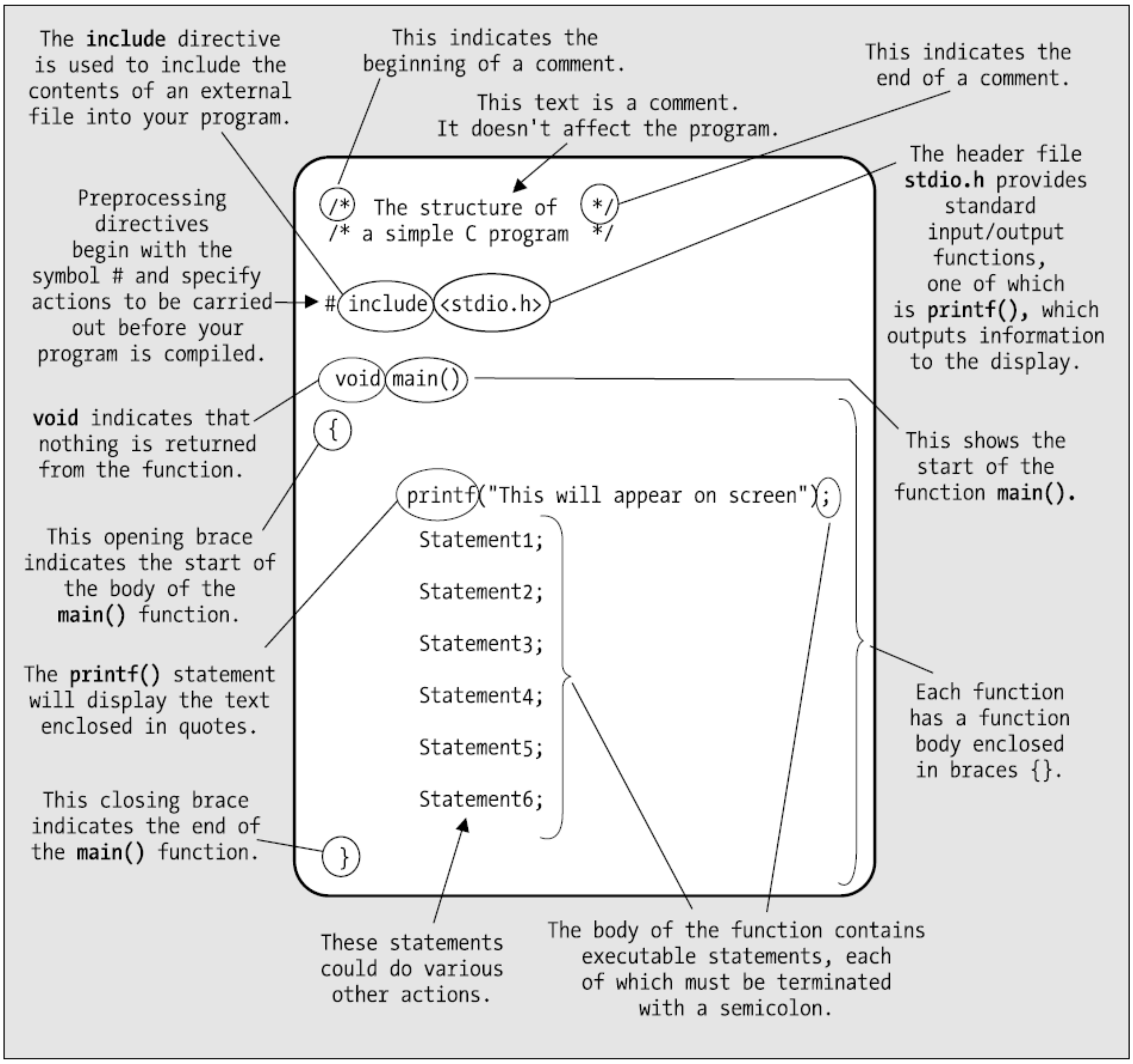


```
// Program: mean_strange.c
#include <stdio.h>
main() { int i, j; int answer;
  i = 12; j = 26; answer = (i + j) / 2;
  printf(
    "The mean of %d and %d is %d\n",
    i,
    j,
    answer
  );
}
```

Listing 5.4 The ‘mean’ program written in a ‘strange’ way

The compiler treats this program just the same as the one shown in Listing 5.3. Keep in mind, however, that if a program is written in this way it is extremely difficult for you to read and understand it. You are therefore advised to write programs that have proper line breaks, indentations and spacing.

In summary, the following figure illustrates the basic elements of a simple C program.



```
/* The structure of */
/* a simple C program */
#include <stdio.h>
void main()
{
    printf("This will appear on screen");
    Statement1;
    Statement2;
    Statement3;
    Statement4;
    Statement5;
    Statement6;
}
```

The include directive is used to include the contents of an external file into your program.
This indicates the beginning of a comment.
This text is a comment. It doesn't affect the program.
This indicates the end of a comment.
The header file stdio.h provides standard input/output functions, one of which is printf(), which outputs information to the display.
Preprocessing directives begin with the symbol # and specify actions to be carried out before your program is compiled.
void indicates that nothing is returned from the function.
This shows the start of the function main().
This opening brace indicates the start of the body of the main() function.
The printf() statement will display the text enclosed in quotes.
Each function has a function body enclosed in braces {}.
This closing brace indicates the end of the main() function.
These statements could do various other actions.
The body of the function contains executable statements, each of which must be terminated with a semicolon.

Figure 5.5 Elements of a simple C program (Horton 2006, Figure 1-4)

Basic elements of C

After studying the examples in the previous subsection, you should find that C programming is really not that difficult. In this section, we'll start to look at the details of various elements of the C language: variables, constants, statements, operators and expressions.

Variables

We can think of a variable as a place for storing a value. You refer to a variable through its name. The place a variable uses to store value is a memory location.

Variable declarations

As discussed in the previous subsection, a variable must be declared before use. The declaration of a variable tells the compiler the data type and name of the variable you are going to use in your program.

The simplest declaration statement consists of the data type and the variable name. Of course, you have to put a semicolon at the end of the statement for termination. The following are two examples:

```
char x;
int answer;
```

Or, you may declare more than one variable, each separated by a comma, in the same line as below:

```
int i, j;
int k, l, m, answer;
```

You may also assign initial values to the variables in the declaration, as in the following examples:

```
char x = '!', y, z;
int i = 1, j = 10, k, answer = 0;
```

It is a matter of style whether you declare one variable or several variables per line. But you should always arrange the declarations in a way that they can be managed easily. For example, the variables for storing the results of some mathematical calculations are declared on the same line (of course, they must be of the same type).

Variable names

You have to pay attention to the following points when using variable names:

- They are made up of letters, digits and underscores.

- The first character must be a letter or an underscore.

- Uppercase and lowercase are distinct — that is, answer, ANSWER and AnSwer are different variable names.

- There can be no spaces and special characters except underscore _ in a variable name.

- The length of the variable name is not restricted, but the compiler takes care of only the first 31 digits of the name. That means if two variables use different names but the first 31 digits of their names are the same, the compiler will treat the two names as referring to the same variable.

- A variable name cannot be the same as keywords in the C language. Examples of keywords are int, float, char, include, and so on; these are part of the syntax of the C language.

Data types

Table 5.1 below summarizes the data types in C. The size of the integer types and floating point types are system-dependent. The char type stores characters in its ASCII code. In the ASCII code set, each character is represented by an integer within the range from 0 to 127. The letter 'a' is represented by 97, 'A' by 65 and digit '0' by 48, digit '1' by 49, and so on. The size of 1 byte can just accommodate the numbers from 0 to 127, that is, the full ASCII code set. You may find that variables of char type participate in mathematical calculations.

Table 5.1 Data types in C (Note: this table as a guide only. Different compilers and different computers may allow different ranges).

Data types	**Descriptions**	**Sizes**	**Typical ranges**
char	Characters	1 byte	–128 to 127
short int	Small integers	Usually 2 bytes	–32768 to 32767
int	Integers	short int ≤ int ≤ long int	–32768 to 32767
long int	Large integers	Usually 4 bytes	–2147483648 to 2147483647
float	Floating point numbers	Usually 4 bytes	–3.4E+38 to 3.4E+38
double	Floating point numbers, with more precision and larger range than float	float ≤ double	–1.7E+308 to 1.7E+308
long double	Floating point numbers, with more precision and larger range than double	double ≤ long double	–1.7E+308 to 1.7E+308

As there is a fixed size for each data type, the range of values that a data type can handle is limited. For example, the `short int` type (assumed to be 2 bytes long) can only handle the integer values from –32768 to 32767. Even for the `long int` type (assumed to be 4 bytes long), there is still a limited range, though it is much larger, from –2,147,483,648 to 2,147,483,647. In the past, you may have thought that a computer could compute any mathematical calculation, no matter how large the numbers are. But you now know that it is not true. A computer also has its limitations.

The data types: `float`, `double` and `long double` are used to store floating point numbers, with the precision increasing from `float` to `long double`. As you've learned, the size of each floating point type is system dependent.

We have talked a lot about data types for numbers. You have learned that the argument for the function `printf` is a *string*; that is, a series of characters. Do you know what data type is used to store a string in C? The answer is an array of the `char` type. You may regard an *array* as a series of variables that are called with the same variable name. We discuss arrays in greater detail in a later section.

The data type of a variable determines what values it can store. Later on, when you write your programs, make sure you have used the appropriate type for the variables.

Constants

There are several types of C constants:

- integer constants;
- floating-point constants;
- character constants; and
- string constants.

Integer constants are whole numbers that do not contain decimal points. Floating-point constants are numbers that contain a fractional portion (a decimal point with an optional value to the right of the decimal point). A character constant is an integer, written as one character within single quotes, such as 'x'. The value of a character constant is the numeric value of the character in the machine's character set. A string constant is a sequence of zero or more characters surrounded by double quotes, e.g. "I am a string". A string constant is an array of characters.

Self-test 5.2

1 What's the difference between an integer variable and a floating point variable?

2 What happens if you assign a number with a decimal to an integer?

Activity 5.3

Write a program with the name `ranges.c` to determine the ranges of `char`, `short int`, and `long` variables, both `signed` and `unsigned`, by printing appropriate values from standard headers and by direct computation.

Operators and expressions

You should now understand what variables are used for, how they are declared, and how the data type of a variable determines what values it can store. You should also be able to perform simple input and output. Without the capability to calculate, your computer would be worth little indeed. In C, operators such as `+`, `-`, `*`, `/` are used to do some kind of operation on a value. This operation might be a calculation or some other operation. The arrangement of different operators represents different expressions.

This section introduces you to C's numerous operators and shows you how different operators can be arranged in different ways to form different expressions. C's operators include:

- arithmetic;
- assignment;
- relational and logical; and
- bitwise and assignment operators.

C supports a wide range of operators, and understanding them is a key to understanding C. Not only must you know how the operators work, you must also understand the proper order that operators execute when they appear together in a single statement.

Arithmetic operators

The basic arithmetic operators are shown in Table 5.2 below. You will find that these operators are usually the same in other programming languages, too.

Table 5.2 Data types in C (Note: this table as a guide only. Different compilers and different computers may allow different ranges).

Arithmetic operators	Descriptions
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus (remainder)

Some points to note when using these operators:

- The `-` operator is also used as the negative sign; for example, `-x + y`.
- For division such as `x / y`, if `x` and `y` are of integer type, then the fractional part will be truncated. That means if `x` is 10 and `y` is 4, the result for the division will be 2. If either `x` or `y` is of the floating point type, the result will be a floating point number, and the result will then become 2.5. But, remember to use a floating-point variable to store the result.
- The modulus operator `%` can be only applied to integer variables.
- You can also refer to Table A.5 on page 1 of Reading 5.3 (Kochan p. 440) for the order of evaluation of the operators.

Assignment operators

As its name implies, an assignment operator assigns value. Table 5.3 shows the assignment operators. The `=` operator is the most basic operator; it assigns a value to a variable.

Table 5.3 Assignment operators

Assignment operators	Examples	Descriptions
`=`	`i = 10`	`i` takes the integer value of 10
`+=`	`i += 10`	`i = i + 10`
`-=`	`i -= 10`	`i = i - 10`
`*=`	`i *= 10`	`i = i * 10`
`/=`	`i /= 10`	`i = i / 10`
`%=`	`i %= 10`	`i = i % 10`
`++`	`++i`	`i = i + 1`
`--`	`--i`	`i = i - 1`

Thus,

```
answer = 10;
```

sets the answer to 10, and

```
x = y;
```

sets `x` to `y`'s value.

The expression `k = k + 5` gets `k`'s old value, adds 5 to it, and stores it back to `k` again. Another way of writing this expression is: `k += 5`

If the constant adding to k is 1, there are four ways of writing the expression:

```
k = k + 1
k += 1
++k
k++
```

The above four statements produce the same effect on `k`. That is, the value of `k` will be incremented by 1 after each of these statements is executed.

The operator `+=` is usually not preferred in writing program statements, as it makes the program less readable than other writing styles. However, you still need to know this operator, as it helps you in reading programs written by others (especially by old-school programmers). Indeed, the `=` operator is just like the `+` or `–` operators. Expressions such as `x = y` can be used anywhere in a mathematical expression such as `x + y` and `x - y`. `x = y` also has a value, just as the value of `x + y` is the result of adding `x` to `y`. The value of `x = y` is the same value as that assigned to `x`; that means the value of `y` here. For example, in

```
z = (x = y)
```

`z` and `x` will have the value of `y` after the statement is executed.

Relational and logical operators

Relational and logical operators are just like arithmetic operators. They take on two values, check according to the tested relation and return 0 if the result is false and 1 if the result is true, e.g.:

```
x > y
```

If `x` takes the integer value 5 and `y` takes the value 10, the above expression is false and returns 0. It should be noted that in C, 0 means false and non-zero integers mean true. Table 5.4 shows the full set of relational and logical operators. With the *same* values for `x` and `y`, the resulting values of the following expressions are:

```
x >= y      // true
x <= y      // true
```

```
x != y      // false
x == y      // true
```

Table 5.4 Relational and logical operators

Assignment operators	Descriptions
`>`	Larger than
`>=`	Larger than or equal to
`<`	Less than
`<=`	Less than or equal to
`==`	Equal to
`!=`	Not equal to
`&&`	And
`||`	Or
`!`	Not

Relational and logical operators are mostly used in control flow statements such as `if`, `while`, `for`, and so on, which are described in the next section. A simple example of the `if` statement is:

```
if (x > y) {
    printf("x is greater than y.\n");
} else {
    printf("x is not greater than y.\n");
}
```

Output of these statements then depends on the value of x and y. For example, if `x` = 5 and `y` = 10, then the output will be:

```
x is not greater than y.
```

If `x` = 10 and `y` = 5, then the output will be:

```
x is greater than y.
```

If the tested condition involves more than more than one expression, the `&&` (and) or `||` (or) operators may be used. For example:

```
if ((x > 100) && (x < 200)) {
  printf("x lies between 100 and 200.\n");
}
```

The above example tests whether the value of `x` lies between 100 and 200. You can write it in this way:

```
if (x > 100 && x < 200) {
  printf("x lies between 100 and 200.\n");
}
```

The brackets make the statement more readable. Note, however, that the following expression is wrong:

```
if (200 > x > 100) {      // syntax error!
  printf(“x lies between 100 and 200.\n”);
}
```

The `&&` operator returns a true value only if both expressions it takes on return true values. The `||` operator returns a true value if either or both expressions it takes on return true values. The `!` operator is a unary operator (An operator such as `+`, which takes two operands, is sometimes called a binary or dyadic operator. The prefix form is known as a unary or monadic operator; that is, it takes a single operand). It turns false to true, and vice versa. For example:

```
if (!(x > y)) {
    printf(“x is smaller than or equal to y.\n”);
} else {
    printf(“x is greater than y.\n”);
}
```

Bitwise operators

C provides six operators for bit manipulation. All the operators shown in Table 5.5, except the one’s complement operator `~`, are binary operators and as such take two operands. Bit operations can be performed on any type of integer value in C — be it `short`, `int`, `long`, `long long`, and `signed` or `unsigned` — and on characters, but cannot be performed on floating-point values.

Table 5.5 Bitwise operators

Bitwise operators	**Descriptions**
`&`	bitwise AND
`|`	bitwise inclusive OR
`^`	bitwise exclusive OR (XOR)
`<<`	left shift
`>>`	right shift
`~`	1’s complement (unary)

The first three operators in Table 5.5, `&`, `|` and `^`, are **bitwise logical operators**. They work on their operands bit by bit starting from the least significant (i.e. the rightmost) bit, setting each bit in the result, as shown in Table 5.6.

Table 5.6 Result of logical bitwise operations

op1	op2	op1 & op2	op1 | op2	op1 ^ op2
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

`<<` and `>>` are bitwise shift operators. They are used to move bit patterns to the left and right respectively. The shift operators are used in the following forms

Left shift: `op << n`
Right shift: `op >> n`

where `op` is the integer expression to be shifted and `n` is the number of bit positions to shift.

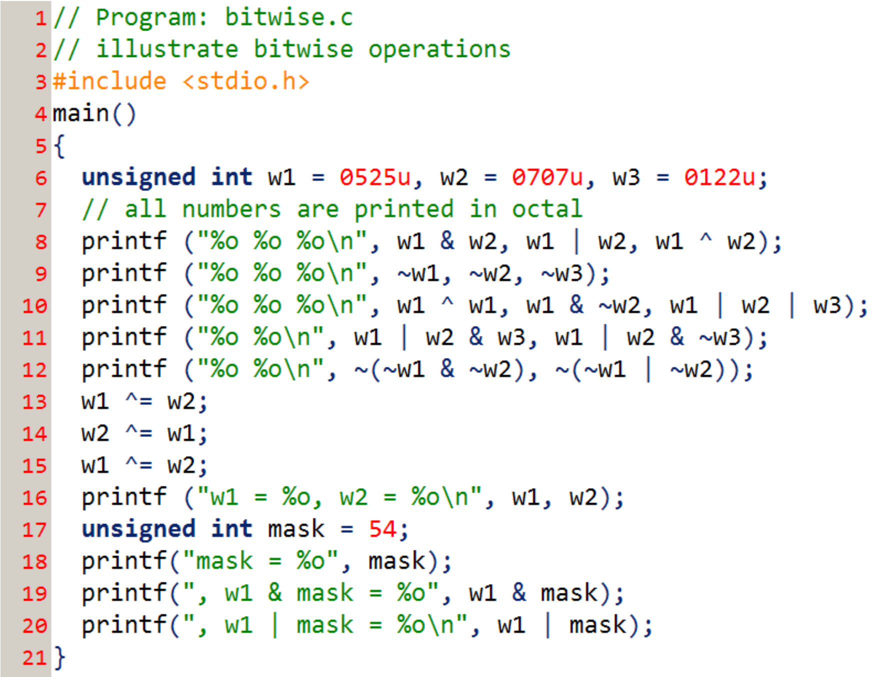
The left-shift operation causes all the bits in the operand `op` to be shifted to left by `n` positions. The leftmost `n` bits in the original bit pattern will be lost and the rightmost `n` bit positions that are vacated to be filled with zeros. Similarly, the right-shift operation causes all the bits in the operand `op` to be shifted to the right by `n` positions. The rightmost `n` bits will be lost. The leftmost `n` bit positions that are vacated will be filled with zeros if `op` is an unsigned integer. If `op` is a signed integer, then the operation is machine-dependent.

Shift operators are often used for multiplication and division by the powers of two. Left-shifting an operand by `n` bits is equivalent to multiplying it by 2^n. Similarly, right-shifting an operand by `n` bits is equivalent to dividing it by 2^n.

The shift operators, when combined with the logical bitwise operators, are useful for extracting desired bits of data from an integer field that holds multiple pieces of information. This process is known as masking. The operand that is used to perform masking is called the mask.

Activity 5.4

Compile and execute the following program.



```
// Program: bitwise.c
// illustrate bitwise operations
#include <stdio.h>
main()
{
  unsigned int w1 = 0525u, w2 = 0707u, w3 = 0122u;
  // all numbers are printed in octal
  printf ("%o %o %o\n", w1 & w2, w1 | w2, w1 ^ w2);
  printf ("%o %o %o\n", ~w1, ~w2, ~w3);
  printf ("%o %o %o\n", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
  printf ("%o %o\n", w1 | w2 & w3, w1 | w2 & ~w3);
  printf ("%o %o\n", ~(~w1 & ~w2), ~(~w1 | ~w2));
  w1 ^= w2;
  w2 ^= w1;
  w1 ^= w2;
  printf ("w1 = %o, w2 = %o\n", w1, w2);
  unsigned int mask = 54;
  printf("mask = %o", mask);
  printf(", w1 & mask = %o", w1 & mask);
  printf(", w1 | mask = %o\n", w1 | mask);
}
```

Listing 5.5 Program to illustrate bitwise operations

Work out each of the operations from the ‘bitwise’ program using pencil and paper to verify that you understand how the results were obtained.

Precedence of operators and order of evaluation

When multiple operators are used in an expression such as `++x * y % z`, which operator should be evaluated first? The answer here is `((++x) * y) % z`. Why is the expression grouped in this way? The language follows the rules of precedence and associativity of the operators.

The following reading summarizes the rules for precedence and associativity of all operators, including some that we have not yet discussed.

Reading 5.3

Kochan, S G (2005) *Programming in C*, 3rd edn, Indianapolis: Sam’s Publishing, 440–42.

Self-test 5.3

1 It the variable x has the value of 10, what are the values of x and a after each of the following statements is executed *separately*?

```
a = x++;
a = ++x;
```

2 To what value does the expression 10 % 3 evaluate?

3 To what value does the expression 5 + 3 × 8 / 2 + 2 evaluate?

Activity 5.5

1 You wrote the program mean3.c for calculating the average of three numbers: 12345, 23456 and 34567 in the Activity 5.2. What data type was used for storing these three numbers and the answer?

Try to use the data type: short int instead of int. Name the new program mean3short.c. Compile and execute the program to see what happens. What is the reason for such a result?

2 Write a program with the name sizes.c to print out the sizes of the data types: char, short int, int, long int, float and double on your Linux system.

Flow control statements

The computer executes the statements inside a program one-by-one in a sequential order. Sometimes, however, we may want the computer to execute a statement repeatedly, or depending on a condition. This has to be done with the help of flow control statements. We now discuss conditional statements such as `if-else` and `switch`, and *loop statements* such as `for` and `while`. Some of these have been informally introduced in previous sections.

The if-else statement

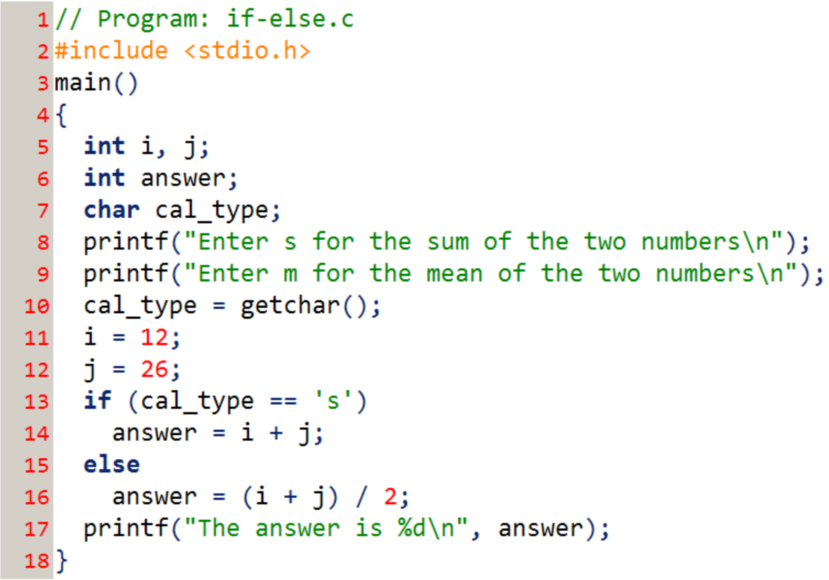
The `if-else` statement executes program statements depending on the truth of a condition. Its basic syntax is:

```
if (conditional_expression)
   <statement>;
```

The statement will be executed if `conditional_expression` is evaluated to be true. Nothing will be done otherwise. You came across some examples when we talked about the relational and logical operators in the last section. `if` is often used with the `else` clause. The syntax is:

```
if (conditional_expression)
   <statement1>;
else
   <statement2>;
```

If `conditional_expression` is true, `statement1` will be executed. Otherwise, `statement2` will be executed. Listing 5.6 shows a program example (`if-else.c`) to illustrate the `if-else` statement.



```
// Program: if-else.c
#include <stdio.h>
main()
{
  int i, j;
  int answer;
  char cal_type;
  printf("Enter s for the sum of the two numbers\n");
  printf("Enter m for the mean of the two numbers\n");
  cal_type = getchar();
  i = 12;
  j = 26;
  if (cal_type == 's')
    answer = i + j;
  else
    answer = (i + j) / 2;
  printf("The answer is %d\n", answer);
}
```

Listing 5.6 A program to illustrate the if-else statement

The following is a sample run of the program:

```
Enter s for the sum of the two numbers.
Enter m for the mean of the two numbers.
s ← your input
The answer is 38.
```

Another sample run:

```
Enter s for the sum of the two numbers.
Enter m for the mean of the two numbers.
m ← your input
The answer is 19.
```

getchar is a library function provided by C. Its information is contained in the header file stdio.h, just like printf. The statement

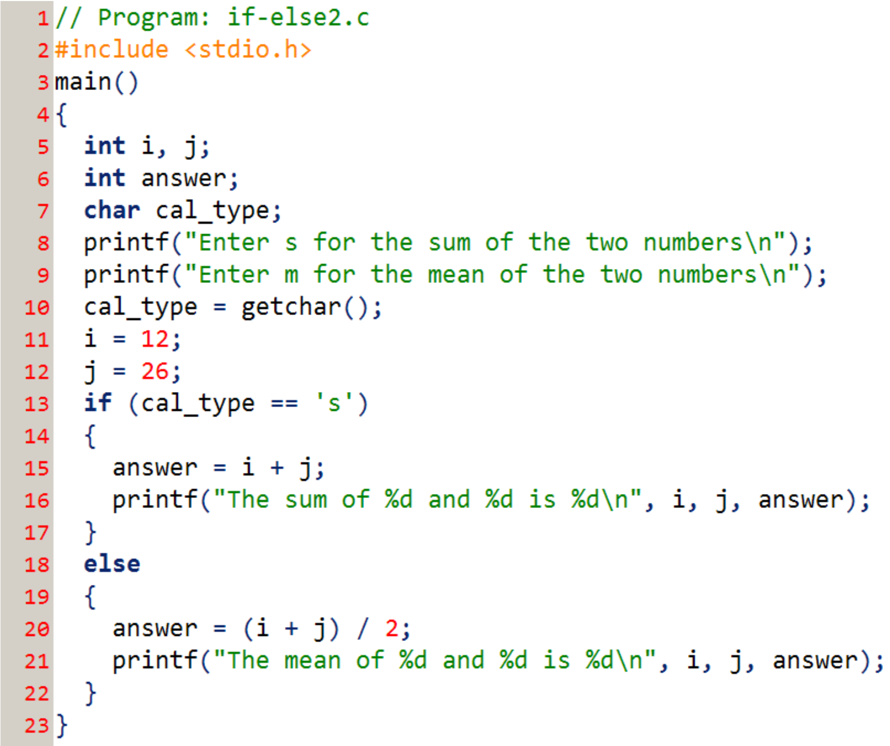
```
getchar(cal_type);
```

reads in a character from the keyboard and stores it in the variable cal_type. The if statement checks the input value. If the input value is 's', the sum of i and j will be calculated; otherwise, the mean of i and j will be calculated.

When more than one statement is to be executed in if-else, you have to group the statements into a compound statement, or a block, by using braces {}. A compound statement means that, although it consists of many statements, it is syntactically equivalent to a single statement, except that there is no semicolon after the {} but there should be semicolons after each statement inside the block. The compound statement can also be applied to statements like switch, for and while (which are introduced later). The corresponding syntax for if-else is:

```
if (condition)
{
  <statement1>;
  <statement2>;
  :
}
else
{
  <statementi>;
  <statementi+1>;
  :
}
```

The following program, ‘if-else2’, is an improved version of ‘if-else’. Statements are grouped into a compound statement `{}`.

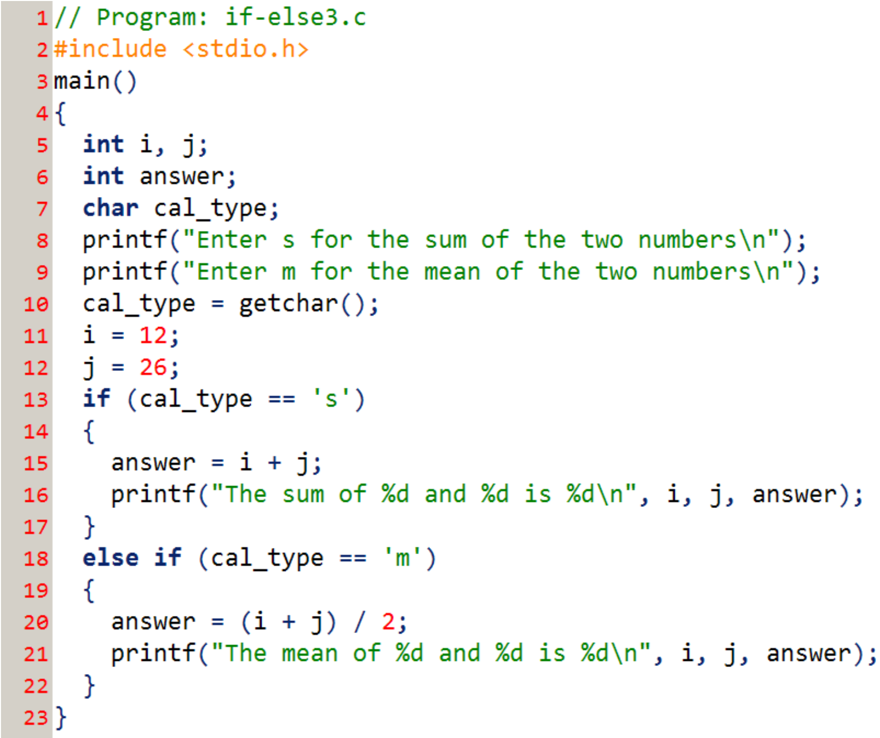


```
// Program: if-else2.c
#include <stdio.h>
main()
{
  int i, j;
  int answer;
  char cal_type;
  printf("Enter s for the sum of the two numbers\n");
  printf("Enter m for the mean of the two numbers\n");
  cal_type = getchar();
  i = 12;
  j = 26;
  if (cal_type == 's')
  {
    answer = i + j;
    printf("The sum of %d and %d is %d\n", i, j, answer);
  }
  else
  {
    answer = (i + j) / 2;
    printf("The mean of %d and %d is %d\n", i, j, answer);
  }
}
```

Listing 5.7 An improved program to illustrate the if-else statement

Can you find any problems with the above two examples? Though the two `printf` statements tell the users to enter ‘`s`’ or ‘`m`’ to choose the calculation type, the `if` statement only checks whether the input value is ‘`s`’ or not. It calculates the mean of the two numbers whenever the input is not ‘`s`’. So if you enter values other than ‘`s`’ and ‘`m`’ — for instance, ‘`h`’ — the mean will still be calculated.

The following listing shows the solution for this problem:



```
// Program: if-else3.c
#include <stdio.h>
main()
{
  int i, j;
  int answer;
  char cal_type;
  printf("Enter s for the sum of the two numbers\n");
  printf("Enter m for the mean of the two numbers\n");
  cal_type = getchar();
  i = 12;
  j = 26;
  if (cal_type == 's')
  {
    answer = i + j;
    printf("The sum of %d and %d is %d\n", i, j, answer);
  }
  else if (cal_type == 'm')
  {
    answer = (i + j) / 2;
    printf("The mean of %d and %d is %d\n", i, j, answer);
  }
}
```

Listing 5.8 Another improved program to illustrate the if-else-if statement

In the above example, we have used another `if` statement in the `else` clause of the first `if` statement to further test the input value. If the input value is not ‘`s`’ or ‘`m`’, the program will produce no output.

You can use the if-else-if-else-if-else... to test for multiple conditions as shown in the following example in Listing 5.8. The indentations for the if-else statements just aim to make the code more readable. Remember that C is a free format language.

```c
// Program: if-else4.c
#include <stdio.h>
main()
{
  int i, j;
  int answer;
  char cal_type;
  printf("Enter s for the sum of the two numbers\n");
  printf("Enter m for the mean of the two numbers\n");
  printf("Enter d for the difference the of two numbers\n");
  printf("Enter p for the product of the two numbers\n");
  cal_type = getchar();
  i = 12;
  j = 26;
  if (cal_type == ’s’)
  {
    answer = i + j;
    printf("The sum of %d and %d is %d\n", i, j, answer);
  }
  else if (cal_type == ’m’)
  {
    answer = (i + j)/2;
    printf("The mean of %d and %d is %d\n", i, j, answer);
  }
  else if (cal_type == ’d’)
  {
    answer = i - j;
    printf("The difference of %d and %d is %d\n", i, j, answer);
  }
  else if (cal_type == ’p’)
  {
    answer = i * j;
    printf("The product of %d and %d is %d\n", i, j, answer);
  }
} /* for main */
```

Listing 5.9 A program to illustrate the if-else-if-else-if-else-if statement

The following is a sample run of the program:

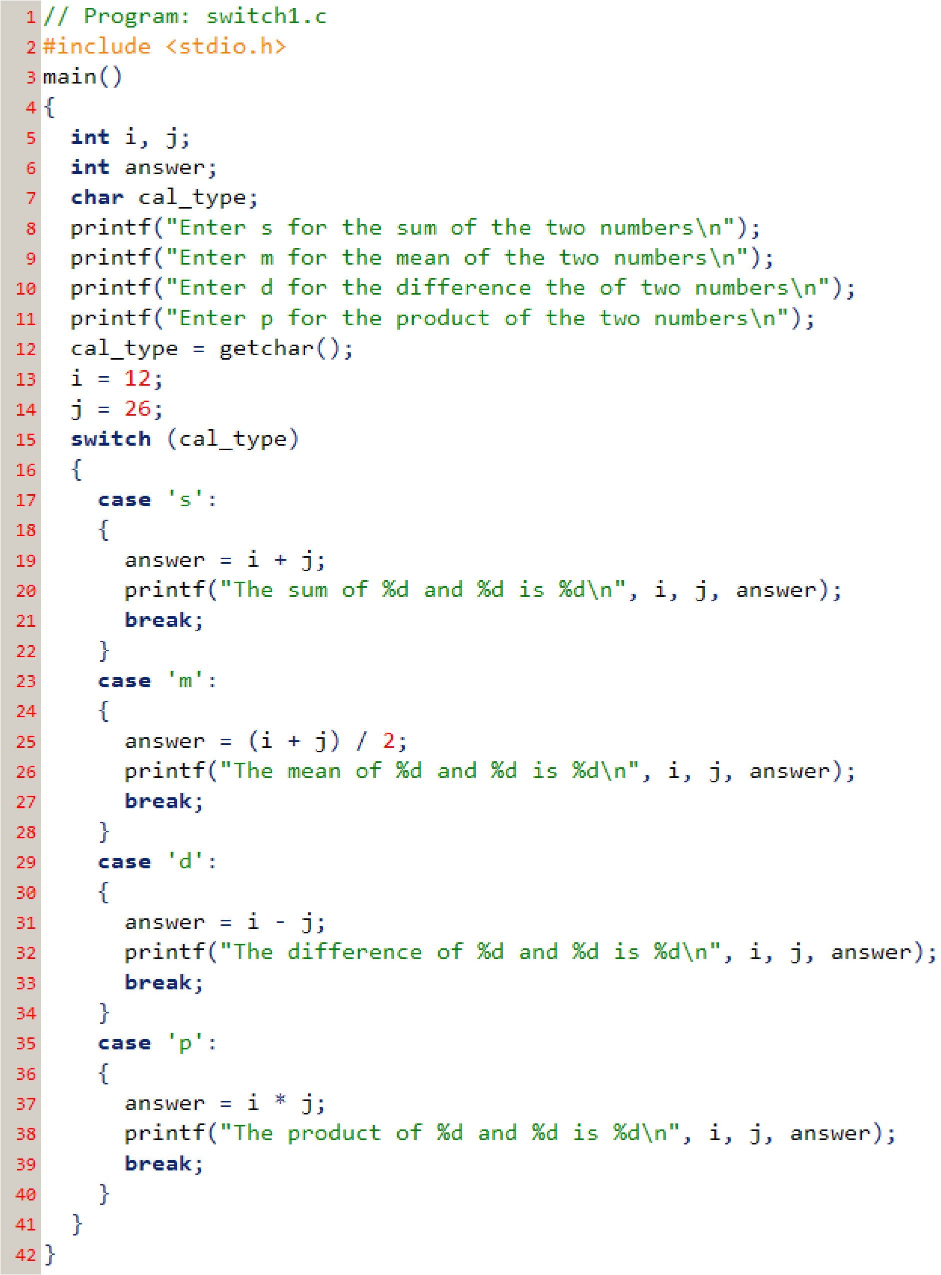
```
Enter s for the sum of the two numbers.
Enter m for the mean of the two numbers.
Enter d for the difference of the two numbers.
Enter p for the product of the two numbers.
d ← your input
The difference of 12 and 26 is -14.
```

Another sample run:

```
Enter s for the sum of the two numbers.
Enter m for the mean of the two numbers.
Enter d for the difference of the two numbers.
Enter p for the product of the two numbers.
m ← your input
The mean of 12 and 26 is 19.
```

The switch statement

If you find it difficult to trace the program logic with the nested ‘`if`’s, you can use the `switch` statement instead. The program ‘switch1’ shown in Listing 5.10 performs the same function as program ‘if-else4’ shown in Listing 5.9, but uses the `switch` statement instead of the `if` statement. The `switch` statement executes different statements, depending on the value of a single variable.



```c
// Program: switch1.c
#include <stdio.h>
main()
{
  int i, j;
  int answer;
  char cal_type;
  printf("Enter s for the sum of the two numbers\n");
  printf("Enter m for the mean of the two numbers\n");
  printf("Enter d for the difference the of two numbers\n");
  printf("Enter p for the product of the two numbers\n");
  cal_type = getchar();
  i = 12;
  j = 26;
  switch (cal_type)
  {
    case 's':
    {
      answer = i + j;
      printf("The sum of %d and %d is %d\n", i, j, answer);
      break;
    }
    case 'm':
    {
      answer = (i + j) / 2;
      printf("The mean of %d and %d is %d\n", i, j, answer);
      break;
    }
    case 'd':
    {
      answer = i - j;
      printf("The difference of %d and %d is %d\n", i, j, answer);
      break;
    }
    case 'p':
    {
      answer = i * j;
      printf("The product of %d and %d is %d\n", i, j, answer);
      break;
    }
  }
}
```

Listing 5.10 A program to illustrate the switch statement

When a case is chosen, the statements inside the block will be executed until a `break` statement is encountered. The `break` statement causes an exit from the `switch`. Though you may find a `break` statement for each case in the above example, this is not a syntax rule. You can try to omit the `break` statement for case '`d`' to see what will happen.

Self-test 5.4

1 Write an `if` statement that assigns the value of `x` to the variable `y` only if `x` is between 1 and 20. Leave `y` unchanged if `x` is not in the range.

2 Find the error of the following code segments. Explain how to correct it.

```
switch(n) {
  case 1:
    printf(“The number is 1\n”);
  case 2:
    printf(“The number is 2\n”);
    break;
  default:
    printf(“The number is not 1 or 2\n”);
    break;
}
```

Activity 5.6

1 Write a program that reads in a character from the keyboard and stores in the `char` variable *x*. If the character is a number — that is, from 0 to 9 — then print the message: ‘The input is the number *x*!’. If the character is a letter from ‘a’ to ‘z’, then print the message: ‘The input is the small letter *x*!’. If the character is a letter from ‘A’ to ‘Z’, then print the message: ‘The input is the capital letter *x*!’. If the character is a value other than the above, print the message ‘The input is the symbol *x*’. Save the program in a file named ‘`char_type.c`’.

The following shows the sample runs of the program:

- Sample run 1:

  ```
  Please enter a letter:
  e ← your input
  The input is the small letter e!
  ```

- Sample run 2:

  ```
  Please enter a letter:
  8 ← your input
  The input is the number 8!
  ```

- Sample run 3:

  ```
  Please enter a letter:
  Z ← your input
  The input is the capital letter Z!
  ```

- Sample run 4:

  ```
  Please enter a letter:
  $ ← your input
  The input is the symbol $!
  ```

Have you used the `if-else` statement, or the `switch` statement in your program? Give a reason for your choice.

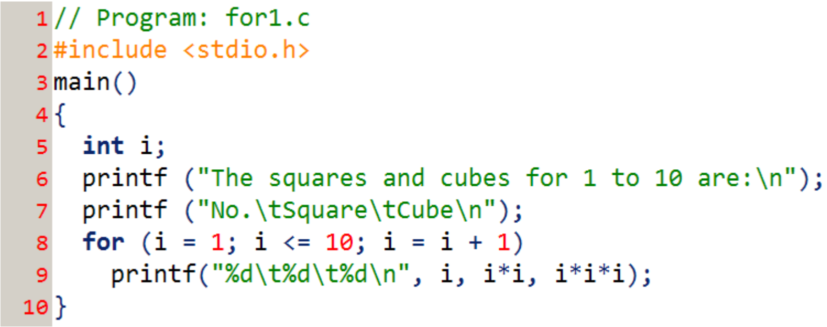
2 The program ‘switch1’ shown in Listing 5.10 did not take into consideration inputs other than ‘`s`’, ‘`d`’, ‘`m`’ and ‘`p`’. Try to modify the program so that when other values are entered from the keyboard, you get the message: ‘Please enter ‘`s`’, ‘`d`’, ‘`m`’ and ‘`p`’!’. Save the new program in a file named ‘`switch2.c`’.

(*Hint*: Use the `default` statement for `switch`.)

Loop statements — `for` and `while`

There are two loop statements that are commonly used in C: `for` and `while`. We’ll examine the `for` statement first.

Listing 5.11 shows an example of a `for` loop that prints out the squares and cubes of the numbers from 1 to 10.



```
// Program: for1.c
#include <stdio.h>
main()
{
  int i;
  printf ("The squares and cubes for 1 to 10 are:\n");
  printf ("No.\tSquare\tCube\n");
  for (i = 1; i <= 10; i = i + 1)
    printf("%d\t%d\t%d\n", i, i*i, i*i*i);
}
```

Listing 5.11 A program to illustrate the for statement

The following shows the sample output of this program:

```
The squares and cubes for 1 to 10 are:
No.     Square  Cube
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

The syntax of the `for` loop is shown below:

```
for (expression1 ; expression2 ; expression3)
    statement;
```

or for multiple statements inside the loop:

```
for (expression1 ; expression2 ; expression3)
{
    statement1;
    statement2;
    :
    statementn;
}
```

The execution of the loop is very often controlled by the value of a variable. Like the example shown in Listing 5.11, the control variable is i. In the for statement, expression1 is an assignment that sets the initial value of the control variable i. For example, i = 1. expression2 is a relational expression that tests the control variable; for example, i <= 10. The loop will stop if this test returns a false value; that is, 0. expression3 is also an assignment that updates the value of the control variable; for example, i = i + 1. The expressions are separated by semicolons. You may omit any one of the three expressions. For example:

```
for ( ; i <= 10 ; i++)
```

and

```
for ( ; i <= 10 ; )
```

You will have to take care of the value of the control variable in other statements, however. As with the one omitting expression1, you may need to set the value of i before entering the for loop; that is:

```
i = 1;
for ( ; i <= 10 ; i++)
```

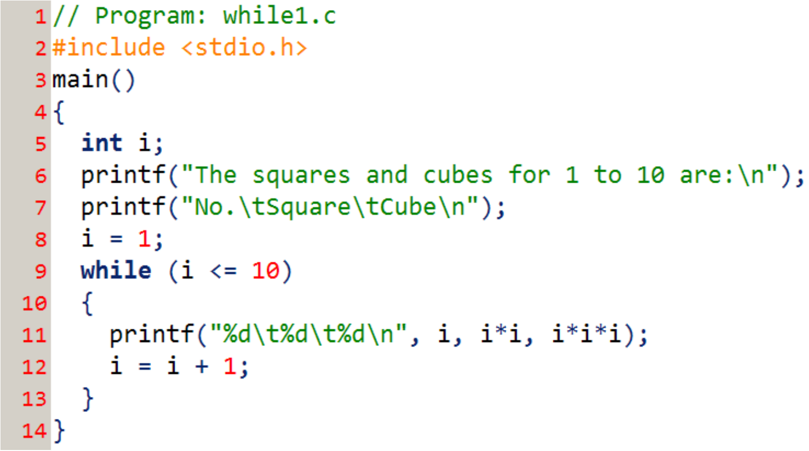
and in the example omitting both expression1 and expression3, you also need to take care of the increment of i; otherwise, the loop may be infinite.

```
i = 1;
for ( ; i <=10 ; )
{
  :
  i++;
  :
}
```

If expression2 is omitted, the loop will go on forever no matter how the control variable is changed inside the loop. The reason is that the test will be taken as permanently true. When this happens, the loop has to be broken by statements such as break and return. These statements are introduced later in the next reading. No matter which expression in the for statement you are going to omit, you have to keep the semicolon. For example, if you omit all three expressions, then the statement should look like:

```
for ( ; ; )
```

The program 'for1' shown in Listing 5.11 may be rewritten using another loop statement, `while`, as shown in Listing 5.12.



```
// Program: while1.c
#include <stdio.h>
main()
{
  int i;
  printf("The squares and cubes for 1 to 10 are:\n");
  printf("No.\tSquare\tCube\n");
  i = 1;
  while (i <= 10)
  {
    printf("%d\t%d\t%d\n", i, i*i, i*i*i);
    i = i + 1;
  }
}
```

Listing 5.12 A program to illustrate the while statement

The basic structure of a `while` loop, as the equivalent to the `for` statement shown above, is shown below:

```
expression1;         /* set up initial value of control
                        variable */
while (expression2)  /* test on control variable */
{
  statement1;
  statement2;
  :
  statementn;
  expression3;       /* update control variable */
}
```

The execution of the `while` loop still depends on the value of a variable (or several variables). Note, however, that only the test of the control variable is included in the while statement. Initializing and updating the control variable is done by ordinary assignment statements. `expression3` does not need to be the last statement of the block. It may be put anywhere inside the block, depending on your program requirement.

In addition to the `for` and `while` loops, there is one loop statement in C that we haven't looked at: the `do-while` statement. The following reading describes the use of the `do-while` statement. You will find that it is very similar to the `while` statement. The `break` and `continue` statements that cause an immediate exit from a loop are also introduced in the reading. After this reading, try the self-test and activity that follow.

Reading 5.4

Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 64–65.

Self-test 5.5

1 When would you use each of the following control statements?

- the `for` statement
- the `while` statement
- the `do-while` statement

2 How many loops can you nest?

Activity 5.7

Write a program that reads in a number and stores it in an `int` variable `i`. Check whether `i` is an odd number or an even number. The program should run continuously until the user enters 0 from the keyboard. Save the program in a file named '`oddeven.c`'.

(*Hint*: Use `scanf("%d", &i);` to read the integer from keyboard into the variable `i`. The `scanf` function is discussed later in the section on input and output.)

The following shows the sample runs of the program:

- Sample run 1:

```
Please enter a number (0 to quit):
100
100 is an even number!
```

- Sample run 2:

```
Please enter a number (0 to quit):
201
201 is an odd number!
```

- Sample run 3:

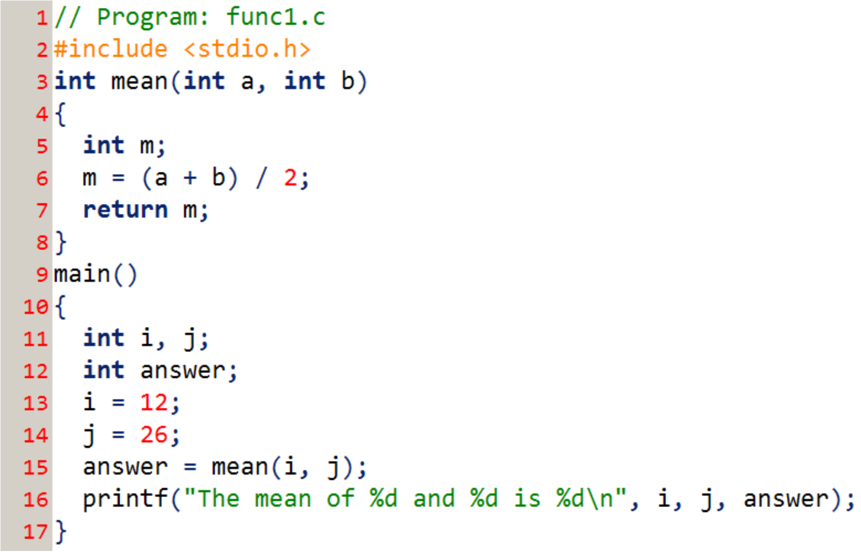
```
Please enter a number (0 to quit):
0
Bye-bye!
```

Functions

You learned to use functions such as `main` and `printf` in previous sections. In this section, we describe in greater detail how functions are used in C. When you write a program that has only several lines, you may not sense the usefulness of functions. If your program contains several thousand lines, however, you will find functions very helpful in constructing a more manageable program structure.

A program example of a function

Before further discussing the benefits of using functions, let's first study an example of a function. This example is shown in Listing 5.13 below.



```
// Program: func1.c
#include <stdio.h>
int mean(int a, int b)
{
  int m;
  m = (a + b) / 2;
  return m;
}
main()
{
  int i, j;
  int answer;
  i = 12;
  j = 26;
  answer = mean(i, j);
  printf("The mean of %d and %d is %d\n", i, j, answer);
}
```

Listing 5.13 A program to illustrate the use of function

The function `mean` in the example computes the mean of two integers passed by the calling statement, and returns the result to it. The definition of the function `mean` is given below. A function must be declared before it can be called in the program, and the definition should be put before the `main` function.

```
int mean(int a, int b)
{
  int m;
  m = (a + b) / 2;
  return m;
}
```

Do you remember what we said about the function `main` in analysing the 'hello, world' program? A function should have a name followed by the brackets `()`. Inside the brackets are the arguments for the function. Do you remember what we mean by an argument? In this example, the name of the function is 'mean', and it has two arguments, `a` and `b`. You also need to specify the data types of the arguments. Here both `a` and `b` are have the `int` type; `a` and `b` are just arbitrary names and are only used in the definition. You should make sure that the writing of `int mean(int a, b)`, instead of `int mean(int a, int b)` is invalid.

Very often the function will return a value to the statement calling it. The data type of the return value is defined by the type specified before the name of the function in the definition. This results in the first statement of the function definition:

```
int mean(int a, int b)
```

↑

Data type for the function.

If the function has no return value, you can use the type `void`. For example, if there were no return value for the function `mean`, the definition would be:

```
void mean(int a, int b)
```

You can omit the data type specified before the function name, just as with `main`. In C, if no return type is specified, the assumed return type is `int`. In the definition, the statements of the function are put inside the braces `{}`. You can also use declared variables for the processing in the function. You should note, however, that these variables cannot be used outside the function that is declared. For example, the variable `m` is declared inside `mean`. If there is a statement inside `main` that refers to `m` such as:

```
m = 10;
```

you will be prompted with a compilation error telling you that the variable `m` is not declared.

The last statement is a `return` statement. It consists of the keyword `return` and a variable (or an expression) containing the return value.

The function definition just defines what the function should do. To use the function, you have to call it in the function `main` (or another function) in your program. In the above example, the calling statement is:

```
answer = mean(i, j);
```

A function is called by its name. You have to pass input values to the function for processing through its arguments. As in the function definition, there must be a bracket () following the function name in which the arguments are put in the calling statement. In our example, the values of integer variables `i` and `j` defined in the `main` function will be passed to the function `mean`. According to the order of the variables inside the `()` in both the function definition and calling statement, `a` and `b` take on the values of `i` and `j` respectively. `a` and `b` are called the formal parameters of the function, whereas `i` and `j` are called the actual parameters. After computation, the `return` statement of the function — that is, `return m;` — passes the value to `mean(i, j)` and the program continues the execution in main . You can treat mean(i, j) as ordinary variables.

Call by value and call by reference

You may ask whether the values of a and b are being changed inside the function mean in the above example, as in the code shown below:

```
int mean(int a, int b)
{
  int m;
  m = (a + b) / 2;
  a = 100;
  b = 200;
  return m;
}
```

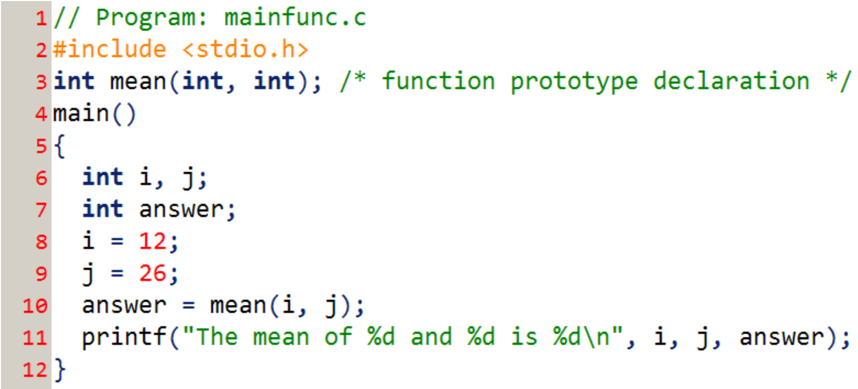
Will the values of i and j be changed accordingly? i and j will not take on the values of a and b after calling the function. The reason is that, in C, formal parameters (that is, a and b in the example) and actual parameters (i and j in the example) are referring to different memory locations. When the function is called, the *values* of the actual parameters are copied to the memory locations referred to by the formal parameters. When the execution of the function ends, the values of the formal parameters are not copied back to the memory locations referred to by the actual parameters. So, although the values of a and b are changed, i and j remain unaffected. In this way, the functions are said to be 'called by value'.

In contrast to this, the functions may be 'called by reference'. In this case, both the formal and actual parameters refer to the same memory locations. In C, this has to be done through pointers, which are introduced later in the unit in the section on arrays and pointers.

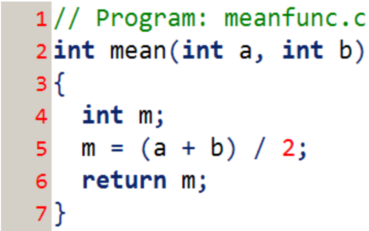
The function prototype declaration

Very often in large programs you break code into functions that perform specific tasks. It is important to lay down the number and data types of the arguments for a function and the data type of the return value. You can then distribute the writing of each function to different programmers. When you are constructing a program that uses all of these functions, you need to focus on what tasks the functions can perform, rather than on the details of the functions. When you use the printf function, for example, you don't necessarily know about the details of the function. However, you know what input should be passed to printf and what the corresponding output will be. In this way, it is not difficult to construct large programs. It is also easier to manage and trace errors. Also, the functions may be reused in different parts of the program. This not only reduces the program's size, but also saves programmers' efforts.

For large programs, it is also common to store function definitions in separate files. But we have said that a function must be defined before the `main` function. How can we do that if the `main` function is stored in one file and the function definitions are in the other files? You may use the function prototype declaration statement. The functions `mean` and `main` stored in Listing 5.13 are stored in separate files as shown in Listing 5.14.



```
// Program: mainfunc.c
#include <stdio.h>
int mean(int, int); /* function prototype declaration */
main()
{
  int i, j;
  int answer;
  i = 12;
  j = 26;
  answer = mean(i, j);
  printf("The mean of %d and %d is %d\n", i, j, answer);
}
```



```
// Program: meanfunc.c
int mean(int a, int b)
{
  int m;
  m = (a + b) / 2;
  return m;
}
```

Listing 5.14 Function prototype declaration example

With the function prototype declaration, the compiler will know that it can find the definition of the function somewhere else. It is the same for the library functions such as `printf`. As mentioned, the `printf` information is stored in the header file `stdio.h`. So although it is not defined in the program, you can still use it. However, what you may find in the header file is actually just the function prototype declaration of `printf`. The definition of `printf` is stored in the system library files, which will be linked with your program at the final stage of the compilation process.

In some programs, you may find the keyword `extern` used in the function prototype declaration, as in the following example:

```
extern int mean(int, int);
```

The keyword `extern` tells the compiler that the function is declared as an external storage class, and that it is optional to a function prototype.

In compiling programs that consist of more than one source file — for example, `prog1.c` and `prog2.c` — just list out all the file names in the command:

```
$ gcc -o myprog prog1.c prog2.c
```

Recursion

The last topic in this section is recursion. Have you thought about whether it is possible to call the function by the function itself? **Recursion** is the name of the term used for calling a function by the function itself. You need to have a very clear mind in order to use recursion effectively, as it is not easy to trace the logic of recursion. Indeed, we do not recommend you use recursion, but you do need to know about it, as there may be times when you need to read other people's programs in which recursion has been used. You can find out more about recursion in the following reading. It explains the term *recursion* and illustrates some examples of its use.

Reading 5.5

Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 86–88.

Self-test 5.6

1 What must be the first line of a function definition? What information does it contain?

2 How many values can a function return?

3 If a function doesn't return a value, what type should it be declared?

Activity 5.8

1 Write a function `double power(double x, int n)` that will compute x^n, the n^{th} power of x. Check to see that it computes $(3.5)^7$ correctly. (The answer is 6433.9296875.)

(*Hint*: Use `print(“%f\n”, power(3.5, 7))` to check the correctness of your function.)

2 Experiment with the following `tbl_of_powers` program:

1	1	1	1	1	1	1
2	4	8	16	32	64	128
3	9	27	81	243	729	2187

How many rows of the table can you compute before the powers that are printed are incorrect? When appropriate, try using the type `double`. Can you get larger numbers this way?

3 The greatest common divisor of two positive integers is the largest integer that is a divisor of both of them. For example, 3 is the greatest common divisor of 6 and 15, and 1 is the greatest common divisor of 15 and 22. Here is a recursive function that computes the greatest common divisor of two positive integers:

```
int gcd(int p, int q)
{
  int r;
  if ((r = p % q) == 0)
    return q;
  else
    return gcd(q, r);
}
```

Write a program to test the function.

Arrays and pointers

So far, we have only introduced simple variables to you. What we mean by *simple* is a variable that is declared with a name and a data type and is able to store a single value at one time. In this section, we introduce you to arrays.

Arrays

Think of an array as a variable that can hold more than one value at the same time. The following is an example of an array declaration:

```
int values[10];
```

This declares an array with the name `values`, consisting of ten elements, each of type `int`. Each element of an array can be treated like a 'simple' variable, but the way you refer to these elements is using the name of an array followed by an integer index inside the brackets `[]`, such as `values[1]`. In C, the index of an array starts from 0. The five elements of `values` are numbered from 0 to 9 and `[1]` is actually referring to the second element of `values`.

The following statements assign values to different elements of the array `values`. The structure of array `values` and the corresponding values after executing those statements are shown in Figure 5.6.

```
values[0] = 197;
values[2] = -100;
values[5] = 350;
values[3] = values[0] + values[5];
values[9] = values[5] / 10;
--values[2];
```

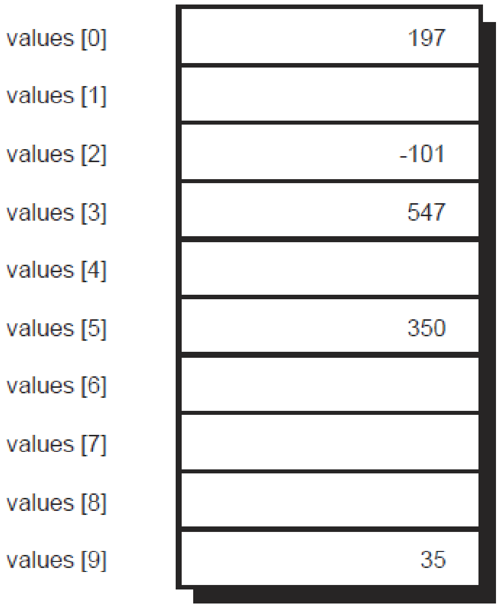


Figure 5.6 The structure of an array (Kochan 2005, Figure 7.2)

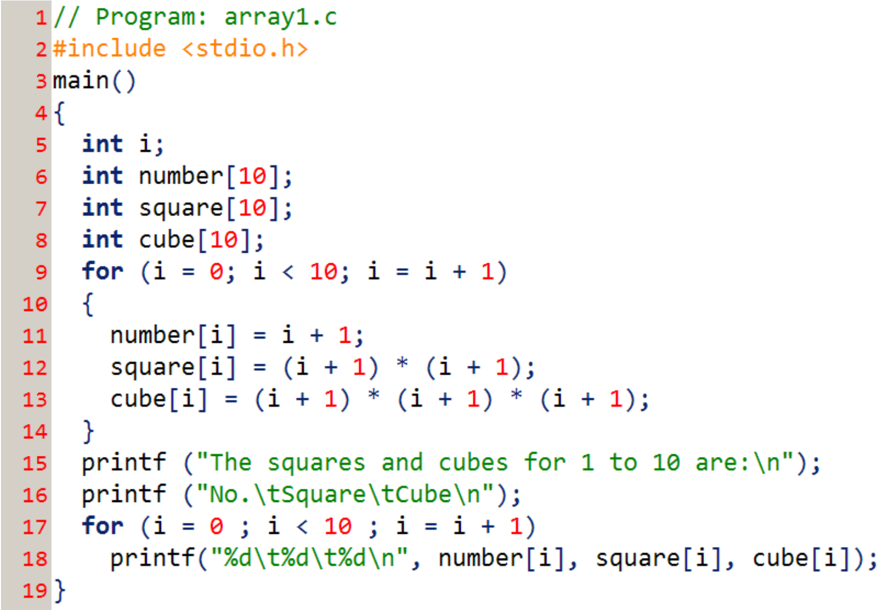
The index does not need to be an integer constant. It may be an integer variable or an integral expression like the following:

```
int i, j;
i = 1;
j = 2;
values[i] = 10;
values[j] = 20;
values[i + j] = 5;
```

This results in `values[1]` holding the value 10, `values[2]` holding the value 20, and `values[3]` holding the value 5. Very often you will initialize all elements of an array to the same value. This can be easily done by a loop. The following code sets all the elements of `values` to 0:

```
int i;
for(i = 0 ; i < 10 ; i = i + 1)
  values[i] = 0;
```

You may ask whether the statement `values = 0;` can initialize all elements of `values` to 0. The answer is that it cannot, and the statement is indeed illegal. The program 'for1' shown in Listing 5.11 has been rewritten using arrays as shown in the following listing:



```
// Program: array1.c
#include <stdio.h>
main()
{
  int i;
  int number[10];
  int square[10];
  int cube[10];
  for (i = 0; i < 10; i = i + 1)
  {
    number[i] = i + 1;
    square[i] = (i + 1) * (i + 1);
    cube[i] = (i + 1) * (i + 1) * (i + 1);
  }
  printf ("The squares and cubes for 1 to 10 are:\n");
  printf ("No.\tSquare\tCube\n");
  for (i = 0 ; i < 10 ; i = i + 1)
    printf("%d\t%d\t%d\n", number[i], square[i], cube[i]);
}
```

Listing 5.15 Program 'for1.c' rewritten using arrays

This example only aims to show you how to use arrays in a program. You may think that the original program (which contains fewer statements) is better. The first `for` loop only performs the calculations of the squares and cubes of the numbers, and the second `for` loop prints the results out. The separation of the computation part and the part for output is important to large programs, as this gives a more manageable program structure.

Arrays can be initialized using the following syntax:

```
int x[5] = {1, 100, 30, 45, 89};
```

Then `x[0]` gets the value of 1, `x[1]` gets the value of 100, and so on. Details of initializing arrays in this way can be found in the following online reading. This reading explains the initialization of some or all elements of an array when the array is defined.

Reading 5.6 (online)

Summit, S (1995–97) *C Programming Notes: Array Initialization*:

http://www.eskimo.com/~scs/cclass/notes/sx4aa.html

In Figure 5.6, we drew the structure of an array as a series of memory locations, each referred to by an integer index. We call this type of array a 'one-dimensional array.'

You may think that there should be two-dimensional arrays and multidimensional arrays. You are right. In C, you may define multidimensional arrays. We discuss them now, starting with two-dimensional arrays. Figure 5.7 shows the declaration of a two-dimensional array and its structure:

```
int z[5][3];
```

	z[][0]	z[][1]	z[][2]
z[0][]			
z[1][]			
z[2][]			
z[3][]			
z[4][]			

Figure 5.7 The structure of a two-dimensional array

A two-dimensional array constructs the structure of a table with the first index referring to the rows and the second index referring to the columns. For further details about multidimensional arrays, you can refer to the following online reading. In this reading, the concept of arrays of arrays (i.e. 'multidimensional' arrays) is discussed. You can use these 'arrays of arrays' for the same sorts of task as we'd use multidimensional arrays for in other computer languages (or matrices in mathematics). Naturally, we are not limited to arrays of arrays, either; we could have an array of arrays of arrays, which would act like a three-dimensional array, etc.

Reading 5.7 (online)

Summit, S (1995–97) *C Programming Notes: Arrays of Arrays ('Multidimensional' Arrays)*:

http://www.eskimo.com/~scs/cclass/notes/sx4ba.html

Pointers

We have introduced the basics of arrays to you, and you may refer to the above reading for further details. Before going any deeper into the topic of arrays, however, we would like to introduce another type of variable to you. It is called a pointer.

You will probably ask what a pointer is. Do you remember we mentioned earlier that a variable is actually a memory location for storing values, and that it is referred to by the variable name? You can think of a pointer as a variable that holds the memory address of a variable. Examples of pointer declaration statements are shown below:

```
int *p;         // pointer to int
int *p1, *p2;   // declaring two pointers to int
char *s;        // pointer to char
float *f;       // pointer to float
```

Can you see the difference between the declaration of simple variables and pointers from this example? It should be obvious: The difference is the asterisk `*`. In the example, `p` is a pointer to integer, `s` is a pointer to a character and `f` is a pointer to `float`. So, you have to specify the type of variable the pointer is pointing to. You may also declare both pointer and simple variable in the same line:

```
int *p, i;    // p: pointer to int, i: integer variable
```

or

```
char c, *s;   // c: char variable, s: pointer to char
```

For the declaration `int *p`, `p` holds the value of a memory address. `*p` holds the value that `p` points to. That means `*p` can be treated as a simple variable. In contrast to the operator `*`, the operator `&` can be used to retrieve the memory address a simple variable. The example in Listing 5.16 shows the relationship of `*` and `&`.

```
// Program: pointer1.c
#include <stdio.h>
main ()
{
  int *p, i = 10;
  int j = 5;
  p = &i;             /* p gets the address of variable i */
  j = *p;             /* j gets the value that p points to */
  printf("Values of j and *p are %d and %d respectively.", j, *p);
}
```

Listing 5.16 A pointer program example

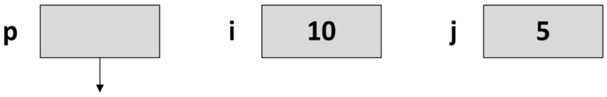
What output do you expect for the above program? It is:

```
Values of j and *p are 10 and 10 respectively.
```

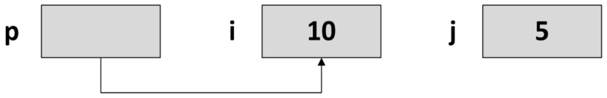
The comments in the program should give you some idea of why the output is what it is. The following pictures should help you have a clearer view of the program.

```
int *p;
int i = 10;
int j = 5;
```

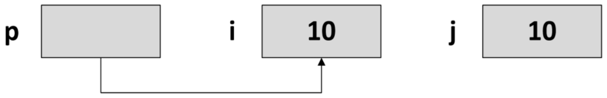
Imagine the pointer as an arrow. The above declaration statements will result in:



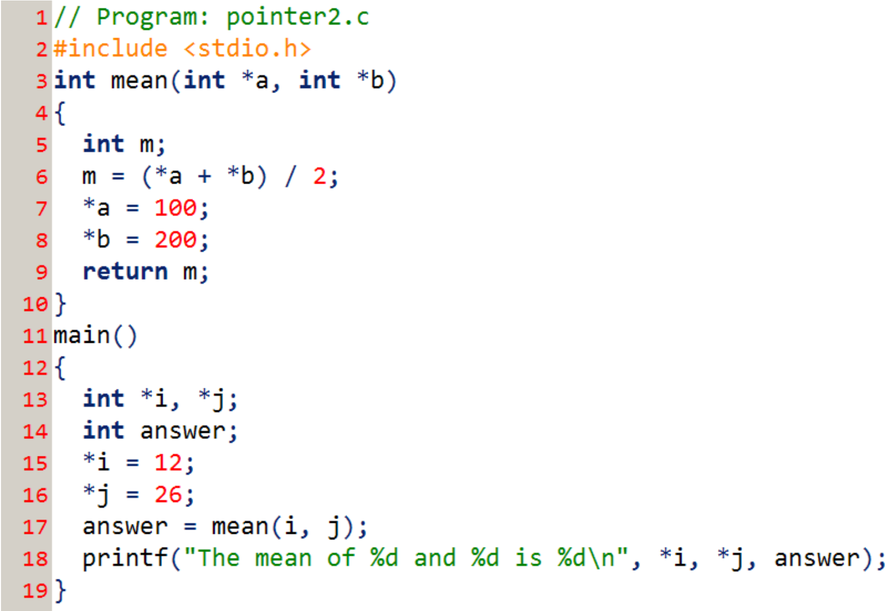
After executing the statement `p = &i;`, `p` gets the address of `i` and `*p` is equivalent to the value of `i`.



Now `*p` holds the value 10. `j` gets the same value as `*p`; through the assignment statement `j = *p`;



Pointers can also be used as function arguments. Remember that we talked about the concept of ‘call by value’ and ‘call by reference’ in the last section. To achieve ‘call by reference’, the function arguments have to be of pointer type. In Listing 5.17, the program ‘func1’ shown in Listing 5.13 has been modified to illustrate this concept.



```
// Program: pointer2.c
#include <stdio.h>
int mean(int *a, int *b)
{
  int m;
  m = (*a + *b) / 2;
  *a = 100;
  *b = 200;
  return m;
}
main()
{
  int *i, *j;
  int answer;
  *i = 12;
  *j = 26;
  answer = mean(i, j);
  printf("The mean of %d and %d is %d\n", *i, *j, answer);
}
```

Listing 5.17 Call by reference example

In the above example, `*i` and `*j` take on the values of `*a` and `*b`, respectively, after calling the function mean. The reason is that, instead of the value of a variable, the address of a memory location is passed to the function through the pointer. So, both the formal and actual parameters are referring to the same memory locations. More examples of ‘call by reference’ are given in the following reading. This reading describes the pointer and function arguments. Pointer arguments enable a function to access and change objects in the function that called it.

Reading 5.8

Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 93–97.

The relationship between pointers and arrays

You should now have a clearer concept of what pointers and arrays are. Indeed, a close relationship exists between pointers and arrays. Operations that can be done with arrays can also be done with pointers. For the following array declaration:

```
int x[10];
```

the first element of the array is `x[0]`. As discussed above, the `&` operator can be used to get the address of a variable and the same operation can be done on an array element, so the address of `x[0]` is given by `&x[0]`. Declaring `y` to be a pointer to `int` is expressed thus:

```
int *y;
```

Setting y to point to array x is expressed by the following:

```
y = &x[0];
```

The above statement can be written as

```
y = x;
```

because x holds the address of the array. To access the other elements of array x, we can now use the pointer y. y points to x[0], so *y gives the value of x[0]. *(y+1) gives the value of x[1], *(y+2) gives the value of x[2], and so on. You should pay attention to the difference between the expressions such as *(y+1) and *y+1. The former gives the value of x[1], and the latter gives the value x[0]+1.

The following reading describes the relationship between arrays and pointers in further detail. It also shows the relationship graphically, which helps you gain a visual understanding of this relationship. In particular, study the two versions of the copyString function (Programs 11.13 and 11.14) carefully.

Reading 5.9

Kochan, S G (2005) *Programming in C*, 3rd edn, Indianapolis: Sam's Publishing, 259–72.

In the following online reading, Steve Summit summarizes some of the similarities of pointers and arrays.

Reading 5.10 (online)

Summit, S (1995–97) *C Programming Notes: 'Equivalence' Between Pointers and Arrays*:

http://www.eskimo.com/~scs/cclass/notes/sx10e.html

Strings

In C, there is no native type to represent character strings. Instead, a character string is represented by an array of type char. A string is terminated by the null character '\0'. A string is also commonly declared and manipulated using a pointer to char. The following shows some examples of declaring strings:

```
char s1[10];
char s2[] = "This is a string";
char *s3;
```

Values may be assigned to the string variables through the assignment statements:

```
*s3 = "Hello!";
s1[1] = 'H';
s1[2] = 'i';
s1[3] = '!';
s1[4] = '\0';
```

The result for the printf statement:

```
printf("Value for s1[] is %s and that for *s3 is %s.",s1, s3);
```

will be

```
Value for s1[] is Hi! and that for *s3 is Hello!.
```

The following reading explains an important difference between the definitions of an array and a pointer. It illustrates more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library — `strcpy` and `strcmp`.

Reading 5.11

Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 104–7.

Command line arguments

In fact, the `main` function has two arguments. The first one is an `int` variable that returns the number of command line argument. The second one is a pointer to a string array that contains the command line arguments. The declaration is as follows:

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

For example, the following command copies file `f1` to file `f2`. The program `cp` in UNIX is used to copy files.

```
cp f1 f2
```

In considering the `main` function of the `cp` program, `argc` takes on the value 3. `argv[0]`, `argv[1]` and `argv[2]` hold the values `"cp"`, `"f1"` and `"f2"` respectively.

In the following reading, Kernighan and Ritchie describe the details of these arguments. You should read through these details, as this is important for understanding the UNIX system calls, which are introduced later on.

Reading 5.12

Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 114–18.

Self-test 5.7

1 What happens if you use an array without initializing it?

2 How many dimensions can an array have?

3 Why is it better to use an array instead of individual variables?

4 How many total elements does the following array have?

int array[2][3][5][8]

Activity 5.9

1 What gets printed in the following statement? Explain your answer.

```
int i = 3, *p = &i;
printf("%d %d %d %d\n", p, *p + 7, **&p, p - (p - 2));
```

2 A *palindrome* is a string that reads the same both forward and backward. Some examples are

```
"ABCBA" "123343321" "otto" "i am ma i" "C"
```

Write a function that takes a string as an argument and returns the `int` value 1 if the string is a palindrome and returns 0 otherwise. How many palindromes can you find in the file `/usr/share/dict/words` on the course Linux server?

3 Modify your palindrome function from question 2 so that blanks and capitals are ignored in the matching process. Under these rules the following are examples of palindromes:

```
"Anna" "A man a plan a canal Panama" "ott o"
```

How many *more* palindromes can you find in the file `/usr/share/dict/words` on the course Linux server?

4 Given the following statement:

```
char *p[2][3] = {"abc", "defg", "hi",
                 "jklmno", "pqrstuvw", "xyz"};
```

Complete the following table:

Expressions	**Equivalent expressions**	**Values**
`***p`	`p[0][0][0]`	`'a'`
`**p[1]`		
`**(p[1] + 2)`		
`*(*(p + 1) + 1)[7]`		`/* error */`
`(*(*(p + 1) + 1))[7]`		
`*(p[1][2] + 2)`		

Structures

A structure is a collection of variables under a single name. The variables of a structure have their own names and data types, and the data type for each variable may be different. A structure is a convenient way of grouping related information in a program. The following shows the declaration of a simple structure:

```
struct course {
  char *code;
  char *name;
  int no_of_students;
}
```

The structure shown above aims at grouping the information of a course. To access the members of the structure, you have to use the following syntax:

```
struct course ct212_apr09;
ct212_apr09.code = "CT212";
ct212_apr09.name = "Network Programming and Design";
ct212_apr09.no_of_students = 160;
```

This means that the member of a structure is accessed through the name of the structure, plus a ‘.’ and the name of the member.

When a pointer to a structure is used, a member of the structure can be accessed using the -> operator. An example is:

```
struct course ct212_apr09, *course_pointer;
ct212_apr09->code = "CT212";
ct212_apr09->name = "Network Programming and Design";
ct212_apr09->no_of_students = 160;
course_pointer = &ct212_apr09;
```

The pointer course_pointer will now point to ct212_apr09, and its members can be accessed using the member access operator ->:

```
course_pointer->code
course_pointer->name
course_pointer->no_of_students
```

Very often, typedef is used to define structures. The above declaration of the structure of a course may be rewritten as follows:

```
typedef struct
{
  char *code;
  char *name;
  int no_of_students;
} course_type;
course_type course;
```

A new data type with the name `course_type` is defined with the `typedef` statement, and the variable course is declared to be of type `course_type`. `typedef` can also be used to define new data types from other simple data type like `int` and `char`.

In the following reading, Kochan describes the details of C structures, and the `typedef` mechanism that allows a type to be explicitly associated with an identifier. Kochan also gives good examples of programs using C structures. You should study this topic carefully because structures are used very often in network programming, as you will learn in the next unit.

Reading 5.13

Kochan, S G (2005) *Programming in C*, 3rd edn, Indianapolis: Sam's Publishing, 166–77, 180–85, 240–43 and 325–27.

Self-test 5.8

1 How is a structure different from an array?

2 What is the structure member operator? What purpose does it serve?

Activity 5.10

Write code that defines a structure named `time`, and which contains three `int` members.

Input and output

We did not go into any detail about input and output in a C program in the previous sections. Indeed, thus far, you have come across only two library functions: getchar and printf for character input and line output, respectively. In this section, we discuss more facilities in C for input from keyboard and output to screen. To use these facilities, the header file stdio.h must be included in your program.

In contrast to getchar, there is a function called putchar for character output on the screen. You have learned the use of getchar. The use of putchar is as simple as getchar, and the following reading gives you further details. getchar and putchar are macros defined in stdio.h that are used to read characters from the keyboard, and to print characters on the screen, respectively. The following reading provides a demonstration program that can be used for reading characters one after another from the standard input file and that then converts each character to lower case and displays it on the screen.

Reading 5.14

Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 151–53.

Often, getchar and putchar will not fit your requirements, as you may need to read or print out a whole line. In such cases you can use the library functions gets and puts.

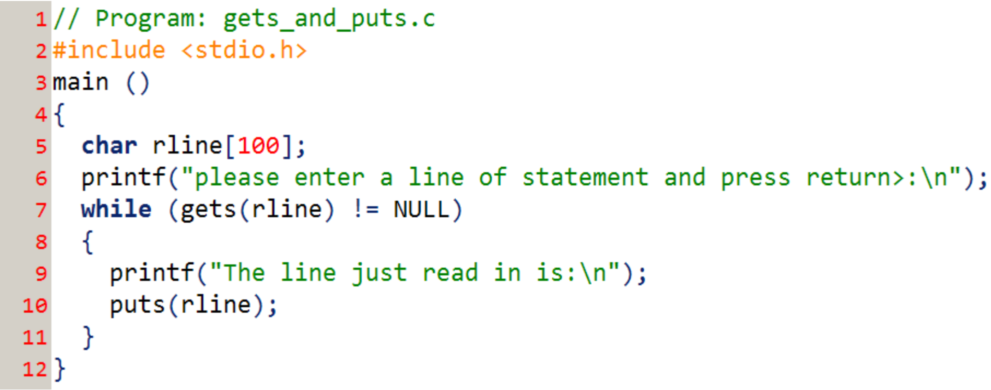
gets reads the whole line from the keyboard until you press <return> or encounter a new line into a string variable. You have to make sure that the string you used to store the input is large enough. The function prototype declaration in stdio.h for gets is as follows:

```
int *gets(char *s);
```

puts does the reverse of gets. It writes a string to the screen and inserts a newline character at the end. Its function prototype declaration is:

```
int *puts(char *s);
```

An example showing the use of both functions is in Listing 5.18.



```c
// Program: gets_and_puts.c
#include <stdio.h>
main ()
{
  char rline[100];
  printf("please enter a line of statement and press return>:\n");
  while (gets(rline) != NULL)
  {
    printf("The line just read in is:\n");
    puts(rline);
  }
}
```

Listing 5.18 Example programs for gets and puts

The following is a sample run of `gets_and_puts`:

```
please enter a line of statement and press <return>:
How do you do?
The line just read in is:
How do you do?
```

In the following reading, Kernighan and Ritchie clearly describe the use of `scanf` and `printf`. These two functions are used for formatted input and output. They are both very important functions. Make sure you master them.

Reading 5.15

Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 153–155 and 157–159.

Self-test 5.9

1 What are the differences between `printf` and `fprintf`?

2 The `scanf` function requires a minimum of two arguments. What are they?

Common programming errors in C

This section examines some of the more common mistakes that a less experienced C programmer could make.

Missing semicolons

Every C statement must end with a semicolon. A missing semicolon may cause considerable confusion to the compiler and result in 'misleading' error messages. Consider the following statements:

```
a = x + y
b = m / n;
```

The compiler will treat the second line as a part of the first line and treat b as a variable that is undefined. As a result, the 'undefined name' error message will be reported for the second line. However, the actual error is the missing semicolon on the first line. As such, both the message and the reported error location are incorrect. When there are no errors in a reported line, you should check the preceding line for a missing semicolon.

Sometimes a missing semicolon might cause the compiler to go 'crazy' and to produce a series of error messages. In such situations, check for a missing semicolon on a line (and the line preceding it) in the beginning of the error list.

Misuse of semicolons

Another common mistake is the erroneous addition of a semicolon in a wrong place. Consider the following code:

```
i = 0;
while (i < 10) ;       // <== erroneous semicolon
{
  printf("%d\n", i);
  i++;
}
```

This code is supposed to print the numbers from 0 to 9 each on a separate line. However, this code will instead print only the number 0 and then exit. This error is caused by the erroneous addition of the semicolon on the `while` line. Since a single semicolon represents a null statement, it therefore is syntactically valid. As such, the compiler does not produce any error message. These kinds of errors are very difficult to detect.

Using = instead of ==

One of the most common mistakes is the confusion between = and ==. The former is an assignment operator that assigns a value into a variable. The latter is a relational operator that compares the equality of two variables in control flow statements. An example is shown below:

```
while (x = 10) {...}
```

The test for the above while loop will be always true, since the assignment statement `x = 10` will have the value of 10 and non-zero integers represent true in C. `x == 10` should be used instead.

Missing break statements in a switch statement

The omission of `break` statements in a `switch` statement may sometimes be intentional, but most of the time it is unintentional.

```
switch (a) {
  case 1:
    printf("a is 1.\n");
  case 2:
    printf("a is 2.\n");
    break;
}
```

The above example will cause both `a is 1.` and `a is 2.` to be printed out if the value of `a` is 1.

Missing braces

You should pay attention to the difference between the nested `if` statements like the following:

```
if (a)
  if (b)
    i = i + 1;
  else
    i = i - 1;
```

and

```
if (a)
  { if (b)
    i = i + 1; }
  else
    i = i - 1;
```

In the former example, the `else` part belongs to `if (b)`. In the latter example, the `else` part belongs to `if (a)`.

In order to avoid confusion and unintentional error, you are strongly recommended to use braces explicitly, even when there is only a single statement inside the `if` or `else` part.

In addition, some programmers prefer to place the braces on separate lines, so that opening and closing braces can be matched easily at a glance.

```
if (a)
{
  if (b)
  {
    i = i + 1;
  }
  else
  {
    i = i - 1;
  }
}
```

Ignoring the order of evaluation of increment and decrement operators

We often use increment or decrement operators in loops. For example:

```
i = 0;
while ((c = getchar()) != '\n')
{
  string[i++] = c;
}
string[i-1] = '\n';
```

The statement `string[i++] = c;` is equivalent to:

```
string[i] = c;
i = i + 1;
```

This is not same as the statement `string[++i] = c;` which is equivalent to:

```
i = i + 1;
string[i] = c;
```

Missing & operator in scanf parameters

All non-pointer variables in a `scanf` call should be preceded by an `&` operator. If the variable `code` declared as an integer, then the statement

```
scanf("%d", code);
```

is wrong. The correct one is

```
scanf("%d", &code);
```

The compiler will not detect these kinds of errors, so be careful!

Using uninitialized pointers

An uninitialized pointer points to garbage. The following code is wrong:

```
main()
{
  int a, *ptr;
  a = 25;
  *ptr = a + 5;
}
```

The pointer `ptr` has not been initialized. The compiler cannot detect these kinds of errors. You are almost certain to get a 'Segmentation fault' when you execute the compiled program.

Missing function prototype declarations

You should also be careful of the danger of not using prototype declarations. C would make an assumption about a function that has not been declared before and thus, although there might be no compilation errors, there would be run-time errors.

The following online reading presents other common mistakes in C. You should read the first three sections carefully, as learning to avoid these errors will help you a lot in writing your C programs. You should also try the exercises provided in the final section of this reading.

Reading 5.16 (online)

Love, T (1996) *ANSI C for Programmers on UNIX Systems: Some Common Mistakes — Miscellaneous*:

http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node33.html

- The **Miscellaneous** link in this reading describes some common general mistakes in C programming:

  http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node34.html

- The **declaration mismatch** link describes errors that result from making the wrong declaration:

  http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node35.html

- The **malloc** link describes errors related to memory allocation and deallocation:

  http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node36.html

- The **Find the bug** link gives you examples of incorrectly written programs and lets you try to find the bugs inside each program:

  http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node37.html

Summary

The C language is usually a full-course topic in its own right, but we have examined it here in only one unit. You may have found, therefore, that the materials did not cover C in great detail. The purpose of this unit, however, is to prepare you for the topics on network programming covered in the next two units. We therefore focused on the particular C programming topics that are necessary for network programming; this is why our discussion of the C language could be compressed into one unit.

In addition to looking at the basics of the C language, we talked about some basic general programming skills. These skills will be helpful for you if you are new to programming. As mentioned repeatedly, *you have to write real programs in order to learn programming*. You should log on to the Linux system specially set up for this course (or the one that you installed on your own PC), and write your own programs, compile them and run them.

The Linux system is the environment for developing your C programs and the network programs in the next two units. In order to become familiar with its operations, you should practice C programming more in the Linux environment. You can refer to *Unit 4* and Lab 1.1 of the Lab Book (Kwan 2009) to revise the basic Linux commands in your programming. This will definitely help you proceed to the topics on network programming in *Units 6* and *7*.

Suggested answers to self-tests and activities

Self-test 5.1

1 The four skills are: attention to detail, stupidity, good memory, and an ability to abstract and to think on several levels.

2 The code is the set of instructions for performing a task, and the data are the set of registers or memory locations that contain the intermediate results used as the program performs its calculations.

Self-test 5.2

1 An integer variable can hold a whole number (a number without a fractional part), and a floating-point variable can hold a real number (a number with a fractional part).

2 You can assign a number with a decimal to an `int` variable. If you're using a constant variable, your compiler probably will give you a warning. The value assigned will have the decimal portion truncated. For example, if you assign 3.14 to an integer variable called `pi`, `pi` will only contain 3. The fractional part, .14, will be chopped off and thrown away.

Self-test 5.3

1 After the first statement, the value of `a` is 10, and the value of `x` is 11. After the second statement, both `a` and `x` have the value 11 (The statements must be executed separately).

2 1

3 19

Self-test 5.4

1 The following code fragment is just one of many possible answers. It checks to see if `x` is greater than or equal to 1 and less than or equal to 20. If these two conditions are met, `x` is assigned to `y`. If these conditions are not met, `x` is not assigned to `y`; therefore, `y` remains the same.

```
if ((x >= 1 && (x <= 20))
    y = x;
```

2 *Error:* there is a missing `break` statement in the statement block for the first `case`.

Correction: Add a `break` statement at the end of the statements for the first `case`. Note that this is not necessarily an error if the programmer wants the statement of `case 2:` to execute every time the `case 1:` statement executes.

Self-test 5.5

1 If you look at the syntax, you can see that any of the three can be used to solve a looping problem. Each has a small twist to what it can do, however. The `for` statement is best when you know that you need to initialize and increment in your loop. If you only have a condition that you want to meet, and you are not dealing with a specific number of loops, `while` is a good choice. If you know that a set of statements needs to be executed at least once, a `do…while` might be best. Because all three can be used for most problems, the best strategy is to learn them all and then evaluate each programming situation to determine which is the best in that context.

2 You can nest as many loops as you want. If your program requires you to nest more than two loops deep, consider using a function instead. You might find sorting through all those braces difficult, so perhaps a function would be easier to follow in code.

Self-test 5.6

1 The first line of a function definition must be the function header. It contains the function's name, its return type, and its parameter list.

2 A function can return either one value or no values.

3 A function that returns nothing should have the type `void`.

Self-test 5.7

1 This mistake does not produce any compilation error. If you do not initialize an array, there can be any value in the array elements, and you might get unpredictable results. As such, you should always initialize variables and arrays so that you know exactly what is in them.

2 You can have as many dimensions as you want as long as your system has enough memory resources.

3 With arrays, you can group like values with a single name.

4 240. This is determined by multiplying 2×3×5×8.

Self-test 5.8

1 The data items in an array must all be of the same type. A structure can contain data items of different types.

2 The structure member operator is a period. It is used to access members of a structure.

Self-test 5.9

1 The two functions printf and fprintf are identical, except that printf always sends output to stdout, whereas fprintf specifies the output stream. fprintf is generally used for output to disk files.

2 The first argument in scanf is a format string that uses special characters to tell scanf how to interpret the input. The second and additional arguments are the addresses of the variable to which the input data are assigned.

Activity 5.2

```
// Program: mean3.c
#include <stdio.h>
main()
{
int i, j, k;
int answer;
i = 12345;
j = 23456;
k = 34567;
answer = (i + j + k) / 3;
printf("The average of the 3 numbers: %d, %d and %d is
    %d\n", i, j, k, answer);
}
```

Activity 5.3

The standard for C specifies that ranges be defined in <limits.h>. The ranges may vary from machine to machine because the sizes for short, int, and long vary for different hardware.

```
// Program: ranges.c
// determines ranges of types
#include <stdio.h>
#include <limits.h>
main ()
{
  /*signed types */
  printf("signed char min = %d\n", SCHAR_MIN);
  printf("signed char max = %d\n", SCHAR_MAX);
  printf("signed short min = %d\n", SHRT_MIN);
  printf("signed short max = %d\n", SHRT_MAX);
  printf("signed int min = %d\n", INT_MIN);
  printf("signed int max = %d\n", INT_MAX);
  printf("signed long min = %ld\n", LONG_MIN);
  printf("signed long max = %ld\n", LONG_MAX);
  /*unsigned types */
  printf("unsigned char max = %u\n", UCHAR_MAX);
  printf("unsigned short max = %u\n", USHRT_MAX);
  printf("unsigned int max = %u\n", UINT_MAX);
  printf("unsigned long max = %lu\n", ULONG_MAX);
}
```

Sample output:

```
signed char min = -128
signed char max = 127
signed short min = -32768
signed short max = 32767
signed int min = -2147483648
signed int max = 2147483647
signed long min = -2147483648
signed long max = 2147483647
unsigned char max = 255
unsigned short max = 65535
unsigned int max = 4294967295
unsigned long max = 4294967295
```

Activity 5.5

1 `short int` is used to store the three numbers, and the answer in `mean3short.c` instead of `int` in `mean3.c` is as shown below:

```
// Program: mean3short.c
#include <stdio.h>
main()
{
  short int i, j, k;
  short int answer;
  i = 12345;
  j = 23456;
  k = 34567;
  answer = (i + j + k) / 3;
  printf("The average of the 3 numbers: %d, %d and
      %d is %d\n", i, j, k,answer);
}
```

Sample output:

```
The average of the 3 numbers: 12345, 23456 and -30969 is 1610
```

The answer for the calculation is wrong. As you can see from the output, the number 34567 has been changed to –30969 after storing in the variable `k` of type `short int`. The reason is that the range for `short int` is from –32767 to 32767. If a number outside this range is stored in the variable of type `short int`, a wrong value results.

2 The program `sizes.c` is shown below:

```
// Program: sizes.c
#include <stdio.h>
main()
{
  printf("size of char: %d byte\n", sizeof(char));
  printf("size short int: %d bytes\n", sizeof(short int));
  printf("size int: %d bytes\n", sizeof(int));
  printf("size long int: %d bytes\n", sizeof(long int));
  printf("size float: %d bytes\n", sizeof(float));
  printf("size double: %d bytes\n", sizeof(double));
}
```

Sample output:

```
size of char: 1 byte
size short int: 2 bytes
size int: 4 bytes
size long int: 4 bytes
size float: 4 bytes
size double: 8 bytes
```

Activity 5.6

1
```
// Program: char_type.c
#include <stdio.h>
main()
{
  char x;
  printf("Please Enter a letter:\n");
  x = getchar();
  if ((x >= '0') && (x <= '9'))
    printf("The input is the number %c!\n", x);
  else if ((x >= 'a') && (x <= 'z'))
    printf("The input is the small letter %c!\n", x);
  else if ((x >= 'A') && (x <= 'Z'))
    printf("The input is the capital letter %c!\n", x);
  else
    printf("The input is the symbol %c!\n", x);
}
```

2
```
// Program: switch1.c
#include <stdio.h>
main()
{
  int i, j;
  int answer;
  char cal_type;
  printf("Enter s for the sum of the two numbers\n");
  printf("Enter m for the mean of the two numbers\n");
  printf("Enter d for the difference the of two numbers\n");
  printf("Enter p for the product of the two numbers\n");
  cal_type = getchar();
  i = 12;
  j = 26;
  switch (cal_type)
  {
    case 's':
      answer = i + j;
      printf("The sum of %d and %d is %d\n", i, j, answer);
      break;
    case 'm':
      answer = (i + j) / 2;
      printf("The mean of %d and %d is %d\n", i, j, answer);
      break;
    case 'd':
      answer = i - j;
      printf("The difference of %d and %d is %d\n", i, j,
             answer);
```

```
        break;
      case 'p':
        answer = i * j;
        printf("The product of %d and %d is %d\n", i, j,
               answer);
        break;
      default:
        printf("Please enter 's', 'd', 'm' or 'p'!");
    } // end of switch
} // end of main
```

Activity 5.7

```
// Program: oddeven.c
#include <stdio.h>
main()
{
  int i;
  int chk_odd;
  printf("Please enter a number (0 to quit):\n");
  scanf("%d", &i);
  while (i != 0)
  {
    chk_odd = i % 2;
    if (chk_odd == 0)
      printf("%d is an even number!\n", i);
    else
      printf("%d is an odd number!\n", i);
    printf("Please enter a number (0 to quit):\n");
    scanf("%d",&i);
  }
  printf("Bye-bye!\n");
}
```

Activity 5.8

1

```
#include <stdio.h>
double power(double x, int n)
{
  int i;
  double answer;
  answer = x;
  for (i = 1 ; i < n ; i = i + 1)
  answer = answer * x;
  return answer;
}
main ()
{
  printf ("%f\n", power(3.5, 7));
}
```

2 If type `int` is used, the number of rows you can compute before the powers are incorrect is 22. If type `double` is used instead, there will be more rows you can compute before the powers are incorrect.

In Activity 5.5 question 2, you computed the size of type `int` and type `double`, which are 4 bytes and 8 bytes, respectively. So a variable of type `double` can obviously hold a number of a much larger value.

3

```
#include <stdio.h>
int gcd (int p, int q)
{
  int r;
  if ((r = p % q) == 0)
    return q;
  else
    return gcd(q,r);
}
main ()
{
  printf("The greatest common divisor of 6 and 15
         is %d\n", gcd(6, 15));
  printf("The greatest common divisor of 15 and 22
         is %d\n", gcd(15, 22));
}
```

Activity 5.9

1 The best way to know the result of the `printf` statement is to perform a real test. The following test program was compiled and run on our system.

```
#include <stdio.h>
main()
{
  int i = 3, *p = &i;
  printf("%d %d %d %d\n", p, *p +7, **&p, p-(p-2));
}
```

When you compiled the program, you may have encountered the following warning message:

```
warning: integer overflow in expression
```

What causes such a warning? You tried to print out the content of `p` as a decimal number that is actually a memory address.

A sample run of the program:

```
-1073742580 10 3 2
```

As mentioned, the content of `p` is a memory address, and you now print it as a decimal number, so the result is –1073742580.

`*p` actually holds the value of `i`; that is, 3. So `*p + 7` results in 10.

`*` and `&` are regarded as the inverse to each other. So `**&p` is equivalent to `*p`, which holds the value of `I`, so the result is 3.

Though p is a pointer to int, it can also participate in mathematical calculations. So, p - (p - 2) results in 2.

2 The following shows the function for checking a palindrome. In writing a program to test the function, remember to include the string.h file, in addition to the stdio.h file. The function prototype declaration for the function strlen is found in the string.h file.

```
int palindrome(char *instr)
{
  int i, j;
  i = 0;
  j = strlen(instr) - 1;
  while (i != j)
  {
    if (instr[i] == instr[j])
    {
      if ((j - i) == 1)
        return 1;
      else
      {
        i++;
        j--;
      }
    }
    else
      return 0;
  }
  return 1;
}
```

3 The following shows the function modified from question 2 to take care of blanks and capital letters in a palindrome. In writing a program to test the function, remember to include the stdlib.h file, in addition to the stdio.h and string.h files. The function prototype declarations for the functions isupper and islower are found in the stdlib.h file.

```
int palindrome1 (char *instr)
{
  int i, j, test1, test2;
  i = 0;
  j = strlen(instr) - 1;
  while (i != j)
  {
    while (instr[i] == ' ')
      {i++;}
    while (instr[j] == ' ')
      {j--;}
    test1 = 0;
    if (isupper(instr[i]) && islower(instr[j]))
      test1 = instr[j] - instr[i];
    test2 = 0;
    if (islower(instr[i]) && isupper(instr[j]))
```

```
      test2 = instr[i] - instr[j];
    if ((instr[i] == instr[j]) ||
        (test1 == 32) ||
        (test2 == 32))
    {
      if ((j-i) == 1)
        return 1;
      else
      {
        i++;
        j--;
      }
    }
    else
      return 0;
  }
  return 1;
}
```

4

Expressions	**Equivalent expressions**	**Values**
`***p`	`p[0][0][0]`	‘a’
`**p[1]`	`p[1][0][0]`	‘j’
`**(p[1] + 2)`	`p[1][2][0]`	‘x’
`*(*(p + 1) + 1)[7]`	`p[1][7][1]`	`/* error */`
`(*(*(p + 1) + 1))[7]`	`p[1][1][7]`	‘w’
`*(p[1][2] + 2)`	`p[1][2][2]`	‘z’

Activity 5.10

```
struct time
{
  int hours;
  int minutes;
  int seconds;
};
```

Glossary

assembler — a computer program that translates a program written in an assembly language into machine code that can be directly executed by the CPU of the targeted platform.

assembly language — a low-level programming language that implements a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. An assembly language is specific to certain physical or virtual computer architecture, and is not portable across different architectures.

C preprocessor — a separate program invoked by the C compiler as the first phase of the compilation process. The preprocessor handles directives for source file inclusion (#include), macro definitions (#define), and conditional inclusion (#if). In some C implementation, the C preprocessor is implemented as part of the compiler program.

compiled language — a programming language whose implementations are typically compilers instead of interpreters. A program translated by a compiler tends to be much faster than an interpreter executing the same program. C, C++, COBOL and FORTRAN are typical compiled languages.

compiler — a computer program that transforms source code written in a computer language into another computer language (i.e. the target language, often having a binary form known as object code). The most common reason for using a compiler is to create an **executable file**.

executable file — also known as an executable program or an executable. An executable program is a file that contains indicated tasks according to encoded instructions.

GNU Compiler Collection (GCC) — a compiler system produced by the GNU Project. Originally named the GNU C Compiler, because it only handled the C programming language, GCC 1.0 was released in 1987. It was later extended to support C++, FORTRAN, Pascal, Java and other programming languages. Nowadays, GCC is the standard compiler on most modern UNIX-like operating systems, including GNU/Linux, BSDs and Mac OS X.

integrated development environment (IDE) — a software application that provides comprehensive facilities to programmers for software development. An IDE normally consists of a source code editor, compiler and/or interpreter, build automation tools, and a debugger. Sometimes (and increasingly often), an IDE also provides version control features and tools to simplify the construction of graphical user interfaces (GUIs).

interpreter — an interpreter is a special program that analyses and executes the source code of a program directly without going through a compilation process. The same source code has to been interpreted every time the program is executed.

interpreted language — an interpreted language is a programming language whose implementations typically take the form of interpreters. A program executed by an interpreter is usually much slower than the executable program created by compiling the same program. Languages that are typically interpreted include JavaScript, Perl, Ruby and PHP.

linker — a computer program that takes one or more object files generated by a compiler and combines them into a single executable program.

object code — an organized collection of named objects; typically these objects are sequences of computer instructions in a machine code format, which may be directly executed by a computer's CPU. Object files are typically produced by a compiler as a result of processing a source code file.

recursion — solving a problem using recursion means the solution depends on solutions to smaller instances of the same problem. In C, a function can call itself recursively to solve some kind of problems — for example, calculating the factorial of a given number.

source code — collection of statements or declarations written in some human-readable programming language. Source code allows the programmer to instruct the computer how to perform predefined tasks. The source code which constitutes a program is usually held in one or more text files called source files.

References

Horton, I (2006) *Beginning C: From Novice to Professional*, 4th edn, Apress.

Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall.

Kochan, S G (2005) *Programming in C*, 3rd edn, Indianapolis: Sam's Publishing.

Kwan, R et al (2009) *A Practical Approach to Internet Programming and Multimedia Technologies*, The Open University Press.

Online references

http://www.alcatel-lucent.com/wps/portal/BellLabs

https://www.bell-labs.com/usr/dmr/www/chist.html

http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node33.html

http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node34.html

http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node35.html

http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node36.html

http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/node37.html

http://www.cprogramming.com

http://www.eskimo.com/~scs/cclass/progintro/top.html

http://www.eskimo.com/~scs/cclass/progintro/sx1.html

http://www.eskimo.com/~scs/cclass/progintro/sx3.html

http://www.eskimo.com/~scs/cclass/progintro/sx4.html

http://www.eskimo.com/~scs/cclass/notes/sx4aa.html

http://www.eskimo.com/~scs/cclass/notes/sx4ba.html

http://gcc.gnu.org