Indira Gandhi
National Open University
School of Sciences

MMT-001
PROGRAMMING AND DATA STRUCTURES

Block

1

INTRODUCTION TO THE C PROGRAMMING LANGUAGE

Curriculum Design Committee

Dr. B.D. Acharya
Dept. of Science & Technology
New Delhi

Prof. Adimurthi
School of Mathematics
TIFR, Bangalore

Prof. Archana Aggarwal
CESP, School of Social Sciences
JNU, New Delhi

Prof. R. B. Bapat
Indian Statistical Institute, New Delhi

Prof. M.C. Bhandari
Dept. of Mathematics
IIT, Kanpur

Prof. R. Bhatia
Indian Statistical Institute, New Delhi

Prof. A. D. Dharmadhikari
Dept. of Statistics
University of Pune

Prof. O.P. Gupta
Dept. of Financial Studies
University of Delhi

Prof. S.D. Joshi
Dept. of Electrical Engineering
IIT, Delhi

Dr. R. K. Khanna
Scientific Analysis Group
DRDO, Delhi

Prof. Susheel Kumar
Dept. of Management Studies
IIT, Delhi

Prof. Veni Madhavan
Scientific Analysis Group
DRDO, Delhi

Prof. J.C. Mishra
Dept. of Mathematics
IIT, Kharagpur

Prof. C. Musili
Dept. of Mathematics and Statistics
University of Hyderabad

Prof. Sankar Pal
ISI, Kolkata

Prof. A.P. Singh
PG Dept. of Mathematics
University of Jammu

Faculty Members
School of Sciences, IGNOU
Dr. Deepika
Prof. Poornima Mital
Dr. Atul Razdan
Prof. Parvin Sinclair
Prof. Sujatha Varma
Dr. S. Venkataraman

Course Design Committee

Prof. C.A. Murthy
ISI, Kolkata

Prof. S.B. Pal
IIT, Kharagpur

Dr. B.S. Panda
IIT, Delhi

Prof. C.E. Veni Madhavan
IISC, Bangalore

Faculty Members
School of Sciences, IGNOU
Dr. Deepika
Prof. Poornima Mital
Dr. Atul Razdan
Prof. Parvin Sinclair
Dr. S. Venkataraman

Block Preparation Team

Prof. Sachin B. Patkar (Editor)
Dept. of Electrical Engineering
IIT, Mumbai

Dr. S. Venkataraman
School of Sciences
IGNOU

This Block is a modified version of Block 2 of CS-04.

December 2007

ISBN-

Further information on the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110 068.

Printed and Published on behalf of Indira Gandhi National Open University, New Delhi, by Director, School of Sciences.

PROGRAMMING AND DATA STRUCTURES

C language was developed by Dennis Ritchie in 1971. The purpose was to port the Unix operating system that was developed by Ken Thompson on a DEC PDP machine to other architectures. C, as described in the book *'The C Programming Language'* by Brian Kernighan and Dennis Ritchie(Englewood Cliffs, N.J.:Prentice-Hall, 1978) was the *de facto* standard for C for several years. The version of C described by this book is known as the **Classic** C or **K & R** C. In 1983, a committee was established to create an ANSI(American National Standards Institute) standard to define the C language. The committee finalised the standard in 1989 and made it available to the general public in 1990. This C is known as ANSI **C**. The ANSI standard was also adopted by ISO(International Standards Organisation), and the resulting standard is called ANSI/ISO **C**. We will refer to this as C89. This is the most widely supported C version.

In 1995, Amendment 1 to the standard was adopted, which among other things added several new library functions. The original 1989 amendment together with Amendment I, is the base document for the standard C++, defining the C subset of C++.

In 1999, a new standard for C was released. This contained many innovative features like variable length arrays. However, not many compilers support this standard. Even gcc, in which all the programs in this course have been tested, does not support all the features of C99.

In this course, we will mostly discuss C89. We will point out some of the differences between C89 and the K & R in this course, but we will give importance to C89. We will also mention some extra C99 features supported by the gcc compiler.

Unlike COBOL, which was developed for business applications, C was not developed with any particular purpose in mind. It is a *general* programming language. In this programme we will be using C programming in a variety of areas like numerical computing, image processing and implementing graph theoretic and number theoretic algorithms.

In this course, we introduce you to programming and data structures through the C programming language. The aim of this course is to give you the C programming background required to carry out the programming tasks in the other courses of the programme.

We hope you are familiar with basics of computers. By basics, we mean that you know how work with files, edit a file in an editor and save your work in a file. If you have never worked with computers, we advice you to learn these basics from any of the many basic books on computers available. In the practical guide, we will explain how to type in a C program, how to save it, how to compile it and how to debug(find and remove the mistakes) in some of the popular packages available for C programming. We have also summarised the basic working of a computer in the appendix. You will not be examined on its contents, but a familiarity with it will help you to understand our discussion in these blocks. We do not expect you to know programming, but if you have any previous programming experience it will certainly be helpful.

The discussion in the book is at a leisurely pace and is intended to be thorough. However, we encourage you to just pick up the essentials of each unit, try to write programs using it and then come back and study the units in greater detail. With little more experience you will be able to appreciate our discussion much better. Initially, you should not worry too much about technicalities like the order of evaluation and associativity of operators. We advice that you use brackets to determine the order and associativity. You can come back to the material and study these details later. However, do not omit these parts!

If you want to go into greater depth, you may refer to the following books.

1. C: The Complete Reference, Fourth Edition, *by H. Schildt*, published by **Tata McGraw-Hill**.
   This book is a good reference for syntax and standard C library functions.
2. A book on C, *Kelley and Pohl* published by **Pearson**.
   You can read this book for a more detailed discussion of topics like dynamic memory allocation. This will also be useful for data structures.
3. The C Programming Language(ANSI C version), by Brian Kernighan and Dennis Ritchie, Second Edition, published by **Prentice-Hall of India**. The classic book on C.

You can send corrections or any other feed back to the course coordinator at the following address:

Dr. S. Venkataraman
School of Sciences
Indira Gandhi National Open University
Maidan Garhi
New Delhi 110 068
email: *svenkat@ignou.ac.in*.

INTRODUCTION TO THE C PROGRAMMING LANGUAGE

Programming a computer means preparing a set of instructions for it to follow. These instructions may be written in one of several high-level (human being-like) languages, such as FORTRAN, COBOL, BASIC, Pascal or C, or in a low-level language, such as the computer's assembly. Eventually the instructions must be translated (or, in the case of assembly, assembled), to produce a machine language program. Since it is the translated version of a program that is actually executed, at this level the language in which the source program was written was immaterial. But some languages are more suited to particular applications than others: for example, FORTRAN has been the language of choice for the creation of programs for scientific or engineering applications: it has a large library of built-in functions that aid in mathematical computation. COBOL has had a distinguished history in the area of programming for commercial or business applications; it was particularly suitable for programmers who did not have a mathematical background. Pascal is the most appropriate language for conveying the concepts of disciplined, structured programming to beginners.

While assembly language is powerful it is unwieldy to use, difficult to learn, and restricts the user to the particular computer for which it is designed. In contrast, languages such as Pascal, FORTRAN and COBOL, while easy to learn, cannot provide such control over the machine's hardware as its assembly can. Thus, a gap existed in the levels of capabilities provided to the programmer by HLLs and LLLs. On the one hand there were the languages that were easy to program in, and that were portable across a variety of platforms; and on the other, there was assembly, difficult to program in, but which could provide vastly more facilities for control, on a generic, particular, machine.

This gap was admirably filled by the advent of C, which offers all of the advantages of the high-level languages, and many of the features that assembly alone can provide. C is not a "large" language, in the sense that FORTRAN and COBOL are. But it has a wealth of operators, and a rich library of built-in functions, that make for ease of programming. In this block, you will find quick introductions to data types, expressions and to many of the operators and functions of C, via several illustrative programs. Because the primary purpose of the programs listed here is pedagogical, they are not very large or complex. On occasion you may find that a program contains statements or operators that haven't been explained before. Unfortunately such "forward references" become necessary if meaningful programs are to be created. But, the use of every operator or function is discussed, if a little briefly when it's first used, then certainly at greater length else where in the block. C is not a difficult language (despite rumours to the contrary), and quite often you can "see" a particular usage as "obvious", without a formal introduction to it. In any case, enough hints and pointers are provided in these "previews" to enable you to proceed ahead without difficulty, and create meaningful programs right away. But if you wish to learn more about a particular topic, you can always refer ahead to the unit where it's discussed, then come back to apply the knowledge that you've gained.

After reading and working through this block consisting of five units, we hope that you will be able to write simple programs in C: Programs involving a variety of data types and expressions and functions for input and output. But it is necessary that you should create, compile, execute and verify the output of every program listed in the text. We cannot over emphasise this: Programming is a skill, just as bicycling, swimming or driving are skills. You can learn it only by practice. No amount of pool-side theorising can teach you swimming. If you wish to learn to swim, at some point you will have to jump into the water; so also reading this (or any text) alone is'nt enough to make you into a expert programmer. The only way out is lots of practice. Happy programming!

UNIT 1 GETTING STARTED

Structure **Page No.**

1.1 INTRODUCTION

In this unit we will introduce you to the C programming language. We begin this unit with an overview of the C programming language, its history and the reasons for its continued popularity in Sec. 1.2.

In Sec. 1.3, we start our discussion of the C programming language with a simple example program. We use the program to tell you how a C program is organised in general.

In Sec. 1.4, we discuss escape sequences which output tab, new line etc. In the last section, we discuss some example C programs to give a feel for the language.

Objectives

After studying this unit, you should be able to

- explain the advantages and disadvantages of C language;
- explain the general structure of C programs; and
- explain the purpose of escape sequences.

1.2 AN OVERVIEW

C is a general purpose computer programming language. It was originally created by Dennis M. Ritchie(circa 1971) for the specific purpose of rewriting much of the **Unix** operating system. This now famous operating system was at that time in its infancy; it was first written for a discarded DEC PDP-7 computer by Ken Thompson, an electrical engineer and a colleague of Ritchie's at Bell Labs. Thompson had written the first Unix in a language he chose to call **B**. **B** itself was based on **BPCL** a language that was developed by Martin Richards, another programmer.

Dennis M. Ritchie

The intention of these programmers was to port **Unix** on to other, architecturally dissimilar machines. Clearly, using any assembly language was out of the question; what they needed was a language that would permit assembly-like (i.e. low or hardware level) operations, such as bit manipulation. At the same time it should be usable on different computers. None of the languages then available served the purpose; they had to create a new one, and C was born. The rest is history; **Unix** became spectacularly successful, in large measure because of its portability. C, in which of most Unix was written, has occupied a pre-eminent position in the development of system programs.

The success enjoyed by **Unix**, and the consequent popularity of C for systems programming, forced it on the attention of applications programmers, who came to

appreciate the rich variety of its **operators** and its **control structures**. These enable, and in fact encourage, the practice of modular programming: The individual sub tasks that make up a large program can be written in independent blocks or modules, each of manageable size. In other words, the language possesses features that make possible a "divide and conquer" approach to large programming tasks. When programs are organised as modules they are necessarily well-planned, because each module contains only the instructions for a well-defined activity. Such programs are therefore more easily comprehensible by other readers than unstructured programs.

Another desirable quality of modular programs is that they may be modified without much difficulty. If changes are required, only a few modules may be affected, leaving the remainder of the program intact. And last but not least, such programs are easier to debug, because errors can be localised to modules, and removed. For a concrete example, suppose that you have to write a program to determine all prime numbers up to ten million which are of the form $k^2 + 1$, where k is an integer.(2, 5, 17 and 37 are some examples of such primes.) Then you can write one module to generate the number $k^2 + 1$ for the next admissible value of k; another module to test whether the last number generated is a prime. Organised in this way, not only will the program be elegant, it would also be easy to debug, understand or modify.

Yet, C is not rigidly structured. The language allows the programmer to forsake the discipline of structured programming, should he or she want to do so.

Another point of difference between C and other languages is in the wealth of its operators. Operators determine the kinds of operation that are possible on data values. The operators of C impart a degree of flexibility and power over machine resources which is unequalled by few other contemporary language, except assembly language.

But assembly language has disadvantages; it is not structured; it is not easy to learn; moreover, assembly programs for any substantive application tend to be too long; and they are not **portable**: An assembly language program written for one computer cannot be executed on another with a different architecture and instruction set. On the other hand, a C program created on one computer can be compiled and run on any other machine that has a C compiler.

Possibly the greatest merit of C lies in its intangible quality which can only be termed elegance. This derives from the versatility of its control structures and power of its operators. In practical terms this often means that C programs can be very short, and it must be admitted, sometimes even cryptic.(Indeed, there is a canard that C stands for cryptic!) While brevity is undeniably a virtue(specially when you consider the vastness of COBOL programs!)there can be too much of it. All too frequently in C programs one comes across craftily constructed statements that take as much time to comprehend, as they must have taken to create. In the interest of other readers therefore(as well as, you may find, in your own)you should resist that temptation to be cryptic that C by its nature seems to encourage.

One common classification of computer language divides them into two broad categories: high-level languages(HLLs) and low-level languages(LLLs). An HLL is a language that is easy for a human beings to learn to program in. A good example is BASIC, with its easily understood statements:

```
10 LET X=45
20 LET Y=56
30 LET Z=X=+Y
40 PRINT X,Y,Z
50 END
```

An LLL is a language that is closely related to machine or assembly language; it allows a programmer the ability to exploit all of the associated computer's resources. This

power however does not come without a price: LLLs are generally difficult to learn, and are usually so closely tied to the architecture of the host machine that even a person who is familiar with one LLL will have to invest a great deal of effort in learning other. Consider the following statement from the MACRO Assembly Language of VAX series of computers manufactured by the Digital Equipment Corporation, USA:

```
POPR#^M<R6,R8,R3>
```

It is highly unlikely that one who is not familiar with VAX's MACRO Assembler will be able to fathom any meaning from this instruction. Moreover, LLL programs are not portable; they cannot be moved across machines with different architectures.

C is a considerable improvement on LLLs: it's easy to learn; it's widely available(from micros to mainframes); it provides almost all the features that assembly does, even bit-level manipulation(while being fairly independent of the underlying hardware). It also has the additional merit that its programs can be concise, elegant and easy to debug. Finally, because there is a single authority for its definition, there is a greater degree of standardisation and uniformity in C compilers than for other HLLs.

It has been often said with some justification that C is the FORTRAN of systems software. Just as FORTRAN compilers liberated programmers from creating programs for specific machines, the development of C has freed them to write system software without having to worry about the architecture of the machine that the software is intended for.(Where architecture-dependent code i.e. assembly code is necessary, it can usually be invoked from within the C environment.)C is a middle level language, not as low-level as assembly, and not as high level as BASIC.

In short, C has all the advantages of assembly language, with none of its disadvantages; and it has all the significant features of modern HLLs. It's an easy language to program in, and makes programming great fun.

As we mentioned in the course introduction, we will discuss ANSI C, also known as C89. We will also mention some extra features provided by C99.

We close this section here. In the next section, we will examine some simple C programs to understand the structure of a C program.

1.3 A C PROGRAM

The best way to learn C or any programming language is to begin writing programs in it; so here's our first C program.(See the practical guide for instructions on how to compile and link programs.)

```
/* Program 1; file name: unit1-prog1.c */
#include<stdio.h>
int main(void)
{
    printf("This is easy!!\n");
    return 0;
}
```

Listing 1: First C program.

There are a few important points to note about this program, points which you will find common to all C programs.

Programs in C consist of **functions**, one of which must be `main()`. When a C program is executed, `main()` is where the action starts. Then other functions may be "invoked". A function is a sub-program that contains instructions or statements to perform a

specific computation on its variables. When its instructions have been executed, the function returns control to the calling program, to which it may optionally be made to return the results of its computations. Because `main()` is a function, too, from which control returns back to the operating system at program termination, in **ANSI C** it is customary, although not required, to include a statement in `main()` which explicitly returns control to the operating environment. Also, `main()` returns an integer value to the operating system. So, we have added the key word `int` before `main()` statement.

C99

The int key word before `main()` is not needed in ANSI C because a function without an explicit return type is assumed to return an integer value. This is not so in C99. If no return type is given, C99 compilers may compile with a warning(as it happens with gcc), but it is not compulsory to do so; the program may also fail to compile.

We'll learn more about functions as we go along, but for now you may recognise them by the presence of parentheses after their names.

Apart from `main()`, another function in the program above is the `printf()`. `printf()` is an example of a "library function". It's provided by the developers of the C library, ready for you to use. C library provides a variety of functions. For example, C library provides many common mathematical functions like the trigonometric functions, exponential function etc. Apart from this, you may use other libraries that provide, for example, graphics functions. These libraries contain the instructions for the function in a compiled form or in the form of object code. So, you do not have to write code in your program for these functions. To enable the use of such libraries, the compilation of C programs is a two part process. In the first part, the program compiles your C file into an object file. The C compiler 'remembers' the functions you have called and the linker combines the translated version of the code you wrote with the object code already found in the library. The second step of the process is called *linking.*

In addition to its variables, a function, including `main()`, may optionally have **arguments**, which are listed in the parentheses following the function name. The arguments of functions are somewhat different from the arguments that people have: they are values of the **parameters** in terms of which the function is defined, and are passed to it by the calling program. The instructions which comprise a function are listed in terms of its parameters and its variables.

In the example above, `main()` has no arguments. `printf()` has one: it's the bunch of characters(more properly: **string**) enclosed in double quotes:

```
"This is easy!!\n"
```

When the `printf()` is executed, its built-in instructions process this argument. The result is that the string is displayed on the output device, which we will usually assume is a terminal. The output of the program will be:

```
This is easy!!
```

with the cursor stationed at the beginning of the next line on the screen. In C a string is not a piece of thread: it's any sequence of characters enclosed in double quotes. A string may not include a "hard" carriage-return (<CR>), i.e. all its characters must be keyed in on a single line on the terminal.

```
"This is most certainly not a C string. <CR>
It contains a carriage return Character."
```

You usually put a carriage return by pressing the 'enter' key.

As we have seen in the case of `printf()`, when a function is invoked, it is passed (the values of) its arguments. It then executes its instructions, using these values. On completion the function returns control to the module from which it was invoked. The nice thing about a function is that it may be called as often as required in program, to

perform the same computational steps on the different sets of values that are passed to it. Thus there can be several invocations to `printf()` within a program, with different string arguments each time. Also, a function can call other functions. For example, in Listing 1 on page 9, the function main() calls the function printf() with the text 'This is easy!!' as the argument. In a program for listing primes of the form $k^2 + 1$ we can have a function that computes $k^2 + 1$ for the next value of k passed to it; another function can test whether the value of $k^2 + 1$ calculated by the first function is a prime or not. These functions could be invoked repeatedly, for different values of k. Functions will be formally introduced in the next Block.

With some compilers you may not require the **preprocessor directive:**

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

which we have written just before `main()` in Listing 1 on page 9. These directives, unlike other C statements, are **not terminated** by a semicolon.

Preprocessing is a phase of compilation that is performed prior to the analysis of the program text. We will have more to say about preprocessor directives later.

`stdio.h` is a **header file** that comes with your C compiler and runtime system and contains data that `printf()` needs in order to output its argument. The #include directive causes the inclusion of external text—in this case the file `stdio.h` —in your sources program before the actual compilation proceeds. We'll have more to say about header files later. If you have problems, consult a resident expert!

Let's go back to Listing 1 on page 9. Observe that the body of the program is enclosed in braces {}; a pair of braces defines a **block**. A program may consist of several blocks, which may include yet other blocks, and so on. The left and right braces which mark a block may be placed anywhere on a line. You will find it easier to read and debug the programs if you align the same column. For the same reason the braces of blocks which are nested inside other blocks are indented a few spaces to the right from the braces of the enclosing block.

```
/* Program 2; file name:unit1-prog2.c  */
#include<stdio.h>
int main()
{
    printf("This is the outermost block.\n");
    {
        printf("This is a nested block.\n");
        {
            printf("This block is nested still more deeply.\n");
            {
                printf("This is the innermost block.\n");
            }
        }
    }
    return (0);
}
```

Listing 2: Block structure.

In Listing 2, note that each left brace is balanced by a corresponding right brace. Blocks contain **statements** such as:

```
printf("This is the innermost block.\n");
```

C statements invariably end with a semicolon. C statements may be of any length and may begin in any column. Each statement may extend over several lines, as long as any included strings are not broken. In C a semicolon by itself constitutes a valid statement, the **null** statement. Null statements are often very handy, and we will quite frequently have occasion to use them.

While writing a C program, you can add comments, which are ignored by the compiler. Comments in programs are included between a backslash-asterisk pair, `/**/`:

```
/* This is a comment about comments.
    Comments may extend over many
  lines, but as a rule they should
            be short.*/
```

Comments are important because they help document a program, and should be written so that its logic becomes transparent. They may be placed wherever the syntax allows "white space" characters: blanks, tabs or newlines.

E1) Do you think comments can be nested, i.e. can you have a comment within a comment? Write a program with a nested comment and see if you can compile it.

In discussion above, we mentioned that a C string cannot contain a carriage return. What should we do if want to break a long string? We have to use an *escape sequence.* Escape sequences are the topic of our discussion in the next section.

1.4 ESCAPE SEQUENCES

You might be wondering about the `\n` (pronounced backslash n) in the string argument of the function `printf()`:

`"This is easy!!\n"`

The `\n` is an example of an **escape sequence:** it's used to print the **newline** character. If you have executed Programs 1 or 2 you will have noticed that the `\n` doesn't appear in the output. Each `\n` in the string argument of a `printf()` cause the cursor to be placed at the beginning of the next line of output. Think of an escape sequence as a "substitute character" for printing hard-to-get characters. In an earlier section we learnt that < CR >s are not allowed in strings; but including `\n`'s within one makes it easy to output it in several lines: if you wish to print a string in two or more lines, place the `\n` wherever you want to insert a new line in the output, as in the example below:

```
/* Program 3; file name: unit1-prog3.c */
void main()
{
    printf("This\nstring\nwill\nbe\nprinted\nin\n9\nlines.\n");
}
```

Listing 3: Line breaks in a C program.

Each `\n` will force the cursor to be positioned at the beginning of the next line on the screen, from where further printing will begin. Here is part of the output from executing the above program:

```
This
string
will
etc ...
```

If a string does not contain a `\n` at its end, the next string to be printed will begin in the same line as the last, at the current position of the cursor, i.e. alongside the first string.

Though < CR >s cannot be included in a string, if you must deal with a long string of characters that cannot fit conveniently in a single line, there is a way of continuing it into the next line. You can insert a backslash(\) followed by a < CR > where you wish to break the string:

```
"This is a rather long string of characters.  It extends \
                    over two lines."
```

We have seen that the "double quotes" character is used to mark the beginning and end of a C string. How can the "double quotes" character itself be included within a string? This time the escape sequence\" comes to our aid: it prints as " in the output. Similarly, the escape sequence \\ helps print \, and \' the single quote character, '. All escape sequences in C begin with a backslash.

Here are some exercises to help you check your understanding of escape sequences.

E2) Give the output of the following program:

```
/* Program 4; file name: unit1-prog4.c  */
#include<stdio.h>
int main()
{
    printf("This is the first line of output.");
    printf("But is this the second\nline of output?");
    return (0);
}
```

E3) Execute the program below and obtain the answer to the question in its printf():

```
/* Program 5; file name: unit1-prog5.c */
#include<stdio.h>
int main()
{
    printf("In how many lines will the output\
 of this program be printed?");
    return (0);
}
```

E4) Give the output of the following programs:

a)
```
/* Program 6; file name: unit1-prog6.c */
#include<stdio.h>
int main()
{
    printf("\"A\\n\",the teacher said,\"is used to\n ");
    printf("insert a new line in a C string.\"");
    printf("\n\"IC,  IC\", said the blind student.\n");
    return (0);
}
```

b)
```
/* Program 7; file name: unit1-prog7.c */
#include<stdio.h>
int main()
{
  printf("To be, or not to be,--");
  printf("that is the question:--\nWhether \'tis ");
  printf("nobler in the mind to suffer\nThe slings ");
  printf("and arrows of outrageous fortune\nOr to ");
  printf("take arms against a sea of troubles,\nAnd ");
  printf("by opposing end them?--To die,--to sleep,--\n");
  return (0);
}
```

E5) Write a C language program that gives for its output:

```
/* This is a C comment.  */
```

E6) Debug the following program:

```
#include[stedio.h]
Main {
}
(print("print this\m" \ *very easy * \).)
```

Be careful to remember that "" is **not** the escape sequence for the "double quote" character (as it is in some versions of BASIC). In C ""is the **null string;** the null string is not "empty", as one might have thought; it contains a single character, the **null character**, ASCII 0 (in which all bits are set to zero). We will see later that the last character of every C string is the (invisible) null character. Its presence helps the computer determine the actual end of the string. Other escape sequences available in C are:

\a	Ring terminal bell (the **a** is for **alert**) [ANSI] extension]
\?	Question mark [ANSI extension]
\b	Backspace
\r	Carriage return
\f	Formfeed
\t	Horizontal tab
\v	Vertical tab
\0	ASCII null character

Placing any of these within a string causes either the indicated action, or the related character to be output. In the next section, we will see some more C programs that will give you a better idea about the structure of C program.

1.5 GETTING A 'FEEL' FOR C

In this section we present a few programs that involve concepts which have not so far been discussed; they'll be covered at a leisurely pace by and by. As you read these programs, you may discover that many of their statements are self-evident; those that may not be are commented. Create these programs on your computer, compile and execute them, and by the time you come to read about the features used in the programs, you shall have a fair idea about them already.

The major change between **ANSI C** and the language described in the first edition of **K & R** is in the declaration and definition of functions. Program 1.11 below is an **ANSI C** compliant program, but it may not run on your machine if you have a non-ANSI compiler. Program 1.12 is the same program modified to run on Classic C.

So far, all the programs that we have written do the same thing every time we run the program; they print some text, the same text every time. But, we would like to write programs that do different things at different times. For example, suppose we want a program that adds two numbers and gives us the answer. How do we do it? For such programs we need programs that use *variables.* As the name suggests, variables store values that can change during the run of the program. The next program uses two values, 5 and 7 and stores them in two variables called x and y. If you want the program to do all the various operations for different integers, say 11 and 12, you can edit the source file, change the values of x and y, compile and run the program again. Go ahead! Try out two three different values for x and y. Notice the int key word before the variables x and y. This indicates that x and y will be used to store integer values.

```
/* Program 8; file name:unit1-prog8.c */
#include<stdio.h>
int main()
{
```

```
    /* Elementary operations with small integers */
    /* C calls small integers ints */
    int x = 5, y = 7, z;
    /* x,y, and z are int variables x is 5,y is 7,
       z doesn't have a value yet; */
    printf("x=%d, y=%d\n", x, y);
    /* each %d prints a decimal integer. */
    z = x - y;
    /* now z is x minus y; */
    printf("z=x-y=%d\n", z);
    z = x * y;
    /*the * means "multiplied by";  */
    printf("z=x*y=%d\n", z);
    z = x / y;
    /* one int divided by another;  */
    printf("z=x/y=%d\n", z);
    z = y / x;
    printf("z=y/x=%d\n", z);
    z = x % y;
    /* guess from the output what
       the % in x % y stands for;  */
    printf("z=x%%y=%d\n", z);
    /* To print percent sign
       we use the escape sequence %% */
    z = y % x;
    printf("z=y%%x=%d\n", z);
    return (0);
}
```

Now, that wasn't very interesting, isn't it? Will it not be better if the programs asks us for values of x and y when it is run? We have given below such a program.

```
/* Program 9; file name: unit1-prog9.c */
#include<stdio.h>
int main()
{
    /* Read values from the keyboard,
       see how scanf () works */
    int x, y, z;
    printf("Enter a value for x.\n");
    printf("Type a samll integer, press<CR>:");
    scanf("%d", &x);
    /* mind that ampersand "&",
       just before x; */
    printf("Enter a value for y:");
    scanf("%d", &y);
    z = x * y;
    printf("z=x*y=%d\n", z);
    return (0);
}
```

```
/* Program 10; file name: unit1-prog10.c */
#include<stdio.h>
int main()
{
    int x, y;
    printf("Enter a value for x:");
    scanf("%d", &x);
    printf("Enter a value for y:");
    scanf("%d", &y);
    /* Is x greater than y? Then say so: */
    if (x > y)
        printf("x is greater than y.\n");
    /* else, if it's not, deny it. */
```

```
    else
        printf("x is not greater than y.\n");
    /* The computer can tell if one
       number is greater than another.*/
    return (0);
}

/* Program 11 file name:unit1-prog11.c */
#include<stdio.h>
int addtwo(int x, int y); /* A function that adds two ints. */
int main()
{
    int val_1, val_2, sum;
    printf("Enter a number:");
    scanf("%d", &val_1);
    printf("Enter another:");
    scanf("%d", &val_2);
    printf("\n\nWill let the function addtwo()\
add them...\n");/*Continued*/
    sum = addtwo(val_1, val_2);
    /* transfer control to addtwo (),
       with arguments val_1 and val_2  */
    printf("\nNow we\'re back in main ()...\
What have we here?\n");/*Continued*/
    printf("\naddtwo () tells us their sum is %d.\n", sum);
    return (0);
}
int addtwo(int p, int q)
{
    printf("\n\nNow I'm reporting from inside addtwo ()...\n");
    printf("\nThe numbers you typed \
did reach here...%d and %d\n", p, q);   /*Continued */
    printf("\ntheir sum is...am working on it...\n");
    return (p + q);
}
```

Here is a C program that solves a quadratic equation using the well known formula. Notice the line

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

in the program. It loads the definitions of maths library functions. We need this line because we are using the `sqrt()` function which gives the square root and `fabs()` that gives the absolute value of a number. This program uses the data type called **float** to store the absolute value of the coefficients and the discriminant because they may be numbers with a decimal point in them. We will discuss **float** date type in the next Unit.

```
/* Program to solve a quadratic equation;*/
/*File name:unit1-quad.c */
#include <math.h>
#include <stdio.h>
int main()
{
    float a, b, c, disc;/*Use decimals.*/
    printf("This program solves the\n the \
quadratic equation ax^2+bx+c=0\n");/*Continued*/
    /*Prompt for the coefficient
    of the second degree term.*/
    printf("Enter the value of a\n");
    /*Read the coefficient of
    the second degree term*/
    scanf("%f", &a);
    /*Stop with an error message if it is 0.*/
    if (a == 0) {
```

```
        printf("Value of a should not be zero. Exiting\n");
        return 0;
    };
    /*If a is not zero, prompt for other
    coefficients and read them in.*/
    printf("Enter the value of b\n");
    scanf("%f", &b);
    printf("Enter the value of c\n");
    scanf("%f", &c);
    printf("You entered a=%f,b=%f,c=%f", a, b, c);
    /*Find the discriminant*/
    disc = b * b - 4 * a * c;
    printf("\n The discriminant is %f\n", disc);
    if (disc == 0) {
        printf("This equation has repeated roots.\n");
        printf("The root is %f with multiplicity 2.", -b / (2 * a));
    }
    if (disc > 0) {
        printf("The roots are real.\n");
        printf("The roots are\n");
        printf("%f", (-b + sqrt(disc)) / (2.0 * a));
        printf("\n and\n");
        printf("%f\n", (-b - sqrt(disc)) / (2.0 * a));
    }
    if (disc < 0) {
        printf("The roots are complex. The roots are\n");
        printf("%f+I*%f", -b / (2.0 * a), sqrt(fabs(disc)) / (2.0 * a));
        printf("\n and\n");
        printf("%f-I*%f\n", -b / (2.0 * a), sqrt(fabs(disc)) / (2.0 * a));
    }
    return 0;
}
```

Here is a C program that performs numerical integration using Simpson's rule. The program computes $\int_0^1 \frac{dx}{1+x^2}$ by Simpson's rule using 4 sub-intervals. Recall Simpson's rule: Let $h = \frac{a-b}{n}$, where a is the lower limit of integration and b is the upper limit of integration and n is the number of subintervals, which should be **even**. If we write $y_i = f(a+ih)$, then

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[y_0 + 4\left(y_1 + y_3 + \cdots + y_{n-1}\right) + 2\left(y_2 + y_4 + \cdots y_{n-2}\right) + y_n\right]$$

Note the definition of the function $\frac{1}{1+x^2}$ here. Since, return can contain expressions, we have done the job of finding the value of $\frac{1}{1+x^2}$ within the return statement. return statement will simply return the value of the expression. We will see more about expressions in Unit 3.

```
/* Simpson's rule; file name:unit1-simpsonf.c*/
#include<stdio.h>
/* function to calculate 1/(1+x*x) */
float f(float x)
{
    return (1 / (1 + x * x));
}

int main()
{
    float integral = 0, h = 1 / 4.0;
    /* Apply Simpson's rule */
    integral =
        (h / 3.0) * (f(0) + 4 * (f(h) + f(3 * h))
              + 2 * f(2 * h)
```

```
                    + f(4 * h));
    printf("The integral is %f\n", integral);
    return (0);
}
```

1.6 SUMMARY

In this unit, we have studied the following points:

1) Because it was written for the purpose of porting Unix system and it was required that C permits low level operations. It also enjoys the advantages of high level languages, being comprehensible. It allows modular programming.

2) The C program is organised into functions and every C program must have a function called main.

3) To print certain special characters we need to use **escape sequences:**

\n	New Line
\'	Single quote '
\"	Double quote "
\\	Backslash \
\a	Ring terminal bell (the **a** is for **alert**) [ANSI] extension]
\?	Question mark [ANSI extension]
\b	Backspace
\r	Carriage return
\f	Formfeed
\t	Horizontal tab
\v	Vertical tab
\0	ASCII null character

4) We also examined some example C programs.

1.7 SOLUTIONS/ANSWERS

E1) No, you cannot have nested comments. This is because the compiler will match the first opening /* with the first closing */ and will interpret the rest of the comment as a part of the program. This will result in an error message from the compiler.

E2) **This is the first line of output.But is this the second line of output?**

E3) **In how many lines will the output of this program be printed?**

E4) a)
```
"A\n",the teacher said,"is used to
insert a new line in a C string."
"IC, IC", said the blind student.
```

b)
```
To be, or not to be,-that is the question:-
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune
Or to take arms against a sea of troubles,
And by opposing end them?-To die,-to sleep,-
```

E5)
```
/*File name:unit1-prog15.c*/
#include<stdio.h>
int main()
```

```
{
    printf("/* This is a C comment */");
    return (0);
}
```

E6) Following are the mistakes in the program:

1) Wrong brackets are used around and the stdio.h is also spelt wrongly. It should be `#include<stdio.h>`.
2) The function name main should not be in capitals. The brackets after main should be round brackets and not flower brackets. The correct way is to type `main ()`.
3) The brackets before and after print statement should be flower brackets and not round brackets. It should be `printf` not `print`. The escape sequence `\m` is not a legal escape sequence.
4) The slash in the comment `\* This is easy*\` is wrong. It should be a backward slash and it should be `/*This is easy*/`.

UNIT 2 DATA TYPES IN C

Structure **Page No.**

2.1 INTRODUCTION

Computer programs work with data. For this data has to be stored in the computer and manipulated through programs. To facilitate this, data is divided into certain specific types. Most modern programming languages are to an extent **typed**; i.e., they admit only of particular, pre-declared types of data or variables in their programs, with restrictions on the kinds of operations permissible on a particular type. Typing helps eliminate errors that might arise in inadvertently operating with unrelated variables. Another benefit of typing is that it helps the compiler allot the appropriate amount of space for each program variable: one byte for a character, two for an integer, four for a real, etc. C too is a typed language, but we shall find that it is not as strongly typed as, for example, Pascal is. Thus, it provides the programmer with a far greater degree of flexibility. At the same time you should not forget that with this greater power inherent in language, the C programmer shoulders all the more responsibility for writing code that shall be robust and reliable in all circumstances.

Interestingly enough, **B** and **BCPL**, the ancestors of **C**, were type less languages.

We start this unit with discussion of the data type int in Sec. 2.2. As you can easily guess, this is used for storing data which are integers like number of students in a class etc. In Sec. 2.3, we discuss the data type char which are used to hold characters. In Sec. 2.4, we will discuss float which are used to hold decimal numbers like decimal approximations to π, $\sqrt{2}$ etc. In Sec. 2.5, we will discuss double. They are also used to store decimal numbers, but they can store larger numbers with greater accuracy. The enum and typedef, which we discuss in sections 2.6 and 2.7, respectively, helps us to create user defined data types.

Objectives

After studying this unit, you should be able to

- state what the basic data types are, how they are declared and how they are used;
- explain the use of modifiers **unsigned**, **double** and **long**;
- create simple MACRO definitions and correctly name the identifiers according to the rules;
- use `scanf()` and `printf()` to read and print variables of different data types; and
- use the typedef statement and explain its purpose.

2.2 VARIABLES OF TYPE int

We have already seen that C uses different data types for storing integers and numbers with a decimal digit. Actually, C program variables and constants are of four main types: char, int, float and double. We start our discussion of data types with the discussion of the data type **int**.

Before an identifier can be used in a **C** program its type must be explicitly **declared**. Here's a **declarative statement** of **C**:

```
int apples;
```

This statement declares the programmer's intent that `apples` will store a signed integer value, i.e., `apples` may be either a positive or a negative integer within the range set for variables of type int. Now this range can be very much machine dependent; it depends, among other things on the **word size** of your machine. For most old DOS compilers for the IBM PC for which the word size is 16 bits `int`s are stored in two consecutive bytes, and are restricted to the range $[-32768, 32767]$. In most of the modern compilers which are 32 bit **ints** are 4-bytes signed integers in the range $[-2147483648, 2147483647]$. In declaring `apples` to be an int you are telling the compiler how much space to allocate for its storage. You have also made an assumption of the range in which you expect its value to lie.

In C it is possible and in fact usual to both declare a variable's type and, where needed, **initialise** its value in the same statement:

```
int salary = 5000;
```

It is **not** correct to assume that a variable which has only been declared e.g.:

```
int volts; /*volts is unpredictable*/
```

but has not been initialised, i.e. assigned a value, automatically gets the value 0.

Let's look at Listing 1, and its output. The program adds two ints x and `y`, and prints their sum, `z`. Recall that the `printf`() can be used to print numbers just as easily as it prints strings. To print an int `x` the following `printf`() will do:

```
printf ("The value of x is: %d\n", x);
```

The `%d` signifies that `x` is to be printed as a decimal integer.

```
/* Program 1; file name:unit2-prog1.c */
#include<stdio.h>
int main()
{
    int x = 5, y = 7, z;
    z = x + y;
    printf("The value of x is: %d\n", x);
    printf("The value of y is: %d\n", y);
    printf("Their sum, z, is: %d\n", z);
    return (0);
}
```

Listing 1: A simple program that uses int.

The output of the program in Listing 1 is appended below.

```
The value of x is:  5

The value of y is:  7

Their sum, z, is:  12
```

To understand the very great importance of using variables in a computation only after having assigned them values, execute the program in Listing 2 on the facing page and determine its output on your computer:

```
/* Program 2; file name:unit2-prog2.c */
#include<stdio.h>
int main()
{
    int x, y, z;                /* x, y and z are undefined. */
    z = x + y;
    printf("The value of x is: %d\n", x);
    printf("The value of y is: %d\n", y);
    printf("Their sum, z is: %d\n", z);
    return (0);
}
```

Listing 2: When variables are not defined.

On our 32-bit linux machine, the output was:

```
The value of x is:  134513628

The value of y is:  -1208279840

Their sum, z is:  -1073766212
```

You may like to compile the program, run it and check the output on your computer.

Now, looking at the output of this program, could you possibly have predicted the values x, y and z would get? **Moral: Never assume a variable has a meaningful value, unless you give it one.**

Try the following exercises now.

E1) Execute the program below, with the indicated arithmetic operations, to determine their output:

```
/* Program 3; file name:unit2-prog3.c */
#include<stdio.h>
int main()
{
    int x = 70, y = 15, z;
    printf("x=%d,\n", x);
    printf("y=%d,\n", y);
    z = x - y;
    printf("Their difference x-y is: %d\n", z);
    z = x * y;
    printf("Their product x*y is: %d\n", z);
    z = x / y;
    printf("The quotient x/y is: %d\n", y);
    return (0);
}
```

Can you explain your results? What do you think is happening here? In particular, determine the values you obtain for the quotient x/y when:

$x = 60, \quad y = 15;$
$x = 70, \quad y = 15;$
$x = 75, \quad y = 15;$
$x = 80, \quad y = 15;$

E2) Execute Program 4 below for the stated values of x and y. What mathematical operation could the % symbol in the statement:

```
z=x%y;
```

signify?

```
/* Program 4; file name: unit2-prog4.c */
#include<stdio.h>
int main()
{
    int x = 60, y = 15, z;
    printf("x=%d,\n", x);
    printf("y=%d,\n", y);
    z = x % y;  /* What does the % operator do? */
    printf("z is: %d\n", z);
    x = 70;
    y = 15;
    z = x % y;
    printf("z is: %d\n", z);
    x = 75;
    y = 15;
    z = x % y;
    printf("z is: %d\n", z);
    x = 15;
    y = 80;
    z = x % y;
    printf("z is: %d\n", z);
    return (0);
}
```

2.2.1 Range Modifiers for int Variables

On occasions you may need to work with strictly non-negative integers, or with integers in a shorter or longer interval than the default for `int`s. The following types of **range modifying** declarations are possible:

unsigned

Usage

unsigned int *variable-name*;
Example: unsigned int `stadium_seats`;

This declaration "liberates" the **sign bit**, and makes the entire word(including the freed sign bit) available for the storage of non-negative integers.

Note

The sign bit–the leftmost bit of a memory word–determines the sign of the contents of the word; when it's set to 1, the value stored in the remaining bits is negative. Most architectures use two's complement arithmetic, in which the sign bit is "weighted", i.e. it has an associated place value which is negative. Thus on a 16-bit machine its value is -2^{15}, or $-32,768$. So a 16-bit signed number such as 10000000 00111111 would have the value $2^0+2^1+2^2+2^3+2^4+2^5-2^{15}=-32,705$. As an unsigned integer this string of bits would have the value 32831.

On PCs the unsigned declaration allows for int variables the range at least [0, 65535] and is useful when one deals with quantities which are known beforehand to be both large and non-negative, e.g. memory addresses, a stadium's seating capacity, etc.

Just as `%d` in the `printf()` prints decimal int values `%u` is used to output unsigned ints, as the program in Listing 3 on the next page illustrates. Execute the program and determine its output; also examine the effect of changing the `%u` format conversion specifiers to `%d` in the `printf()`s. Can you explain your results?

```
/* Program 5; file name:unit2-prog5.c */
#include<stdio.h>
int main()
{
    unsigned int stadium_seats, tickets_sold, standing_viewers;
    stadium_seats = 40000;
    tickets_sold = 50000;
    standing_viewers = tickets_sold - stadium_seats;
    printf("Tickets sold: %u\n", tickets_sold);
    printf("Seats available: %u\n", stadium_seats);
    printf("There could be a stampede because\n");
    printf("there may be nearly %u standees \
at the match.\n", standing_viewers);/*Continued*/
    return (0);
}
```

Listing 3: Example to show the use of unsigned modifier.

Usage

short int *variable-name*
Example: short int friends;

The short int declaration may be useful in instances where an integer variable is known beforehand to be small. The declaration above ensures that the range of `friends` will not exceed that of ints, but on some computers the range may be shorter (e.g. -128 through 127); `friends` may be accommodated in a byte, thus saving memory. There was a time in the olden days of computing, when main memory was an expensive resource, that programmers tried by such declarations and other stratagems to optimise core usage to the extent possible. But for present-day PCs, with memory cheap and plentiful, most compiler writers make no distinction between ints and short ints.)

The `%d` specification in the `printf()` is also used to output short ints.

unsigned short

Usage

unsigned short int *variable-name*;
Example: unsigned short int books;

The range of books will not exceed that of unsigned ints; it may be shorter.

long

Usage

long int *variable-name*;
Example: long int stars_in_galaxy;

This declaration is required when you need to use integers larger than the default for ints. On most computers long ints are 4-byte integers ranging over the interval $[-2147483648, 2147483647]$. When a long integer constant is assigned a value the letter `L` (or `l`) must be written immediately after the rightmost digit:

```
long int big_num = 1234567890L;
```

`%ld` in the `printf()` outputs long decimal **int**s, as you may verify by executing the program in Listing 4:

```
/* Program 6; file name:unit1-prog6.c */
#include<stdio.h>
int main()
```

```
{
    long int population_2020 = 424967295L;
    printf("The population of this country in 2020 AD\n");
    printf("will exceed %ld if we do\n", population_2020);
    printf("not take remedial steps now.\n");
    return (0);
}
```

Listing 4: Format modifier for printing long integers.

unsigned long

Usage

unsigned long int *variable-name*;
Example: usage: **unsigned long int** population_2020;

The **unsigned long** declaration transforms the range of **long int**s to the set of 4-byte non-negative integers. Here population_2020 is allowed to range over [0, 4294967295], (Let's hope that larger-sized words will not be required to store this value!) **unsigned long**s are output by the %lu format conversion specifier in the printf().

In the above declarations shorter forms are allowed: thus you may write:

```
unsigned letters;       /* instead of unsigned int letters; */
long rope;              /* instead of long int rope;*/
unsigned short note;    /* instead of unsigned short int note; */
```

etc.

E3) State the output from the execution of the following C program:

```
/* Program 7; file name:unit1-prog7.c */
#include<stdio.h>
int main()
{
    unsigned cheque = 54321;
    long time = 1234567890L;    /*seconds */
    printf("I\'ve waited a long\
time (%ld seconds)\n", time);/*Continued*/
    printf("for my cheque (for Rs.%u/-), \
and now\n", cheque);/*Continued*/
    printf("I find it\'s unsigned!\n");
    return (0);
}
```

Modify this program appropriately to convert the time in seconds to a value of days, hours, minutes and seconds so that it gives the following additional output on execution:

```
That was a wait of:
                    14288 days,
                     23 hours,
                     31 minutes
                   and 30 seconds.
```

(**Hint:** If a and b are int variables, then a/b is the quotient and a % b the remainder after division of a by b if b is less than a.)

As remarked above, on most current compilers for the PC the range of shorts is not different from the ints; similarly the range of unsigned shorts is the same as that of unsigned ints.

Thus we have:

More than one variable can be declared in a single statement:

```
int apples, pears, oranges, etc;
/* declares apples, pears, oranges and etc to be  ints */
unsigned long radishes, carrots, lotus_stems;
/* radishes, carrots and  lotus_stems are unsigned longs */
```

Consider the statement:

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

This declares `x` to be an `int` with value 1, `y` to be an `int` with value 2, and `z` to be an `int` of unpredictable value. Similarly, the statement:

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

declares `x, y` and `z` to be `int`s. `x` and `y` are undefined. `z` has the value 1.

Octal (base 8) or hexadecimal (base 16) integers may also be assigned to `int` variables; octal and hexadecimal integers are used by assembly language programmers as shorthand for the sequences of binary digits that represent memory addresses or the actual contents of memory locations. (That C provides the facility for computing with octal and hexadecimal integers is a reminder of its original raison d'etre: systems programming). An octal integer is prefaced by a `0` (zero), a hexadecimal by the letters `0x` or `0X`, for example

```
int p = 012345, q = 0x1234; /*p is in octal notation, q is in hex */
long octal_num = 012345670L, hex_num = 0X7BCDEF89L;
```

The `printf()` can be used to output octal and hexadecimal integers just as easily as it prints decimal integers: `%0` prints octal `int`s, while `%x` (or `%X` prints hexadecimal `int`s; `long` octal or hex `int`s are printed using `%lo` and `%lx`, respectively, `%#o` and `%#x` (or `%#X`) cause octal and hexadecimal values in the output to be preceded by 0 and by 0x (or 0X) respectively. (A lowercase x outputs the alphabetical hex digits in lowercase, an uppercase X outputs them in uppercase characters.) Precisely how format control may be specified in a `printf`() is described in the next Unit.

What are the decimal values of `p, q, octal_num` and `hex_num`?

One easy way of verifying the answer to the last question is to get `printf`() to print the numbers in decimal, octal and hexadecimal notation:

```
printf ("As an octal number, hex_num = %l0\n", x);
printf ("As a decimal number, octal_num = %ld\n", x);
printf ("As a hexadecimal number, octal\num = %lx\n", x);
```

What happens if in the course of a computation an `int` or `int` like variable gets a value that lies outside the range for the type? C compilers give no warning. The computation proceeds unhampered. Its result is most certainly wrong. This is in contrast to Pascal where overflow causes the program to be aborted. The program in Listing 5 on the next page of exercise. 5 demonstrates this. There is also an exercise to test your understanding of format modifiers for printing hexadecimal and octal numbers. Please try them.

E4) Print the values of `p, q, octal_num` and `hex_num` using each of the %#0, %#l0, %#lx and %#lX format conversion characters respectively.

E5) Execute the program in Listing 5 on the next page(in which a variable x is multiplied by itself several times), and determine its output on your machine:

```
/* Program 8; file name: unit2-prog8.c */
#include <stdio.h>
int main()
{
    int x = 5;
    printf("x = %d\n", x);
    x = x * x;                  /* now x is 5*5 =25 */
    printf("x = %d\n", x);
    x = x * x;                  /* now x = 25*25 = 625 */
    printf("x = %d\n", x);
    x = x * x;                  /* now x exceeds
                the limit of int on Old 16 bit compilers. */
    printf("x = %d\n", x);
    x = x * x;                  /*now x exceeds
                 the limit of int in 32 bit machines also */
    printf("x = %d\n", x);
    return (0);
}
```

Listing 5: Example demonstrating the effects of overflow.

One statement that often confuses novice programmers is:

```
x = x*x;
```

If you have studied algebra, your immediate reaction may well be: "This can't be right, unless x is 0 or x is 1; and x is neither 0 nor 1 in the program.!" True; we respect your observation; however, the statement:

```
x = x*x;
```

is **not** an equation of algebra! It's an instruction to the computer, which in English translates to the following:

Replace `x` by `x` times `x`.

Or, more colloquially, after its execution:

(new value of `x`) is (old value of `x`)*(old value of `x`).

That's why in Program 8, exercise. 5 on the preceding page, x begins with the value 5, then is replaced by 5*5 (which is (25) then by 25*25 (which is 625), and then by 625*625, which is too large a value to fit inside 16 bits.

In ANSI C, a decimal integer constant is treated as an `unsigned long` if its magnitude exceeds that of signed `long`. An octal or hexadecimal integer that exceeds the limit of `int` is taken to be `unsigned`; if it exceeds this limit, it is taken to be `long`; and if it exceeds this limit it is treated as `unsigned long`. An integer constant is regarded as `unsigned` if its value is followed by the letter u or U, e.g.`0x9999u`; it is regarded as `unsigned long` if its value is followed by u or U and 1 or L, e.g. `0xffffffful`.

The system file `limits.h` available in ANSI C compliant compilers contains the upper and lower limits of integer types. You may #include it before `main`() precisely as your `#include <stdio.h>`:

```
#include<limits.h>
```

and thereby give to your program access to the constants defined in it. For example, the declaration:

```
long largest = LONG_MAX;
```

Table 1: Limits for `int` types.

CHAR_BIT	bits in a char	8
CHAR_MAX	maximum value of char	UCHAR_MAX or SHCAR_MAX
CHAR_MIN	minimum value of char	0 or SCHAR_MIN
INT_MAX	maximum value of int	32767
INT_MIN	minimum value of int	-32767
LONG_MAX	maximum value of long	2147483647
LONG_MIN	minimum value of long	-2147483647
SCHAR_MAX	maximum value of signed char	127
SCHAR_MIN	minimum value of signed char	-127
SHRT_MAX	maximum value of short	32767
SHRT_MIN	minimum value of short	-32767
UCHAR_MAX	maximum value of unsigned char	255
UINT_MAX	maximum value of unsigned int	65535
ULONG_MAX	maximum value of unsigned long	4294967295
USHRT_MAX	maximum value of unsigned short	65535

will initialise `largest` to 2147483647. The values stated below are accepted minimum magnitudes. Larger values are permitted: Let us now write a program that uses `limits.h`. The program in Listing 6 prints the maximum and minimum values of `int` and the maximum value of `unsigned int`.(What is the minimum value of `unsigned int`?)

```
/*Program to print limits; file name: printlimits.c*/
#include<stdio.h>
#include<limits.h>
int main()
{
    printf("Maximum value of int is %d: \n", INT_MAX);
    printf("Minimum value of int is %d: \n", INT_MIN);
    printf("Maximum value of unsigned is %u: \n", UINT_MAX);
    return (0);
}
```

Listing 6: Program that prints limits.

Here is an exercise for you to check for yourself whether you can use the values in limits.h file.

E6) Modify the program so that it prints the limits for the other types in Table. 1. Compile and run the program on your computer and see what are the values you get.

You will agree that the programs we've been working with would be much more fun if we could supply them values we wished to compute with, **while they're executing**, rather than fix the values once and for all within the programs themselves, as we've been doing. What we want, in short, is to make the computer **scan** values for variables from the keyboard. When a value is entered, it's assigned to a designed variable. This is the value used for the variable in any subsequent computation. Program 9 below illustrates how this is done. It accepts two numbers that you type in, and adds them. It uses the `scanf()` function, the counterpart of the `printf`() for the input of data. We have already seen an example in Unit 1, that uses `scanf()` function. Here we will give some more details about this function. We will discuss both this function and `printf()` in greater detail later. But, note one important difference: when `scanf()` is to read a variable `x`, we write

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

In `scanf()`, in contrast to `printf()`, the variable name is preceded by the ampersand, &. In the next Unit we will learn that placing the & character before a variable's name yields the memory address of the variable. Quite reasonably a variable that holds a memory address is called a **pointer**. It points to where that variable can be found. Variables which can store pointers are called **pointer variables**. Where `printf()` uses variable names, the `scanf()` function uses pointers.

Experiment with the program in Listing 7 to see what happens if any of `x` or `y` or `z` exceed the limit of `int`s for your computer.

```
/* Program 9; file name:unit2-prog9.c */
#include<stdio.h>
int main()
{
    int x, y, z;
    printf("Type in a value for int x, and press <CR>:");
    scanf("%d", &x);
    printf("\nx is: %d\n\n", x);
    printf("Type in a value for int y, and press <CR>:");
    scanf("%d", &y);
    printf("\ny is: %d\n\n", y);
    z = x + y;
    printf("The sum of x and y is %d\n", z);
    return (0);
}
```

Listing 7: Experimenting with limits of `int`s.

E7) Write a program that reads two non-negative integers of type **`unsigned long`** and checks if their sum entered exceeds the limit for **`unsigned long`**.

We end this section here. In the next section we will discuss another type of variable that can be used to store characters.

2.3 VARIABLES OF TYPE `char`

Character variables are used to store single characters for the ASCII set. They're accommodated in a single byte. Character variables are declared and defined as in the statement below:

```
char bee = 'b', see = 'c', ccc;
```

This declaratory statement assigns the **character constant 'b'** to the `char` variable `bee`, and the character constant `'c'` to the `char` variable named `see`. The `char` variable named see. The `char` variable `ccc` is undefined.

Character constants are single characters. They must be enclosed in right single quotes. Escape sequences may be also assigned to character variables in their usual backslash notation, with the "compound character" enclosed in right single quotes. Thus the statement:

```
char nextline = '\n';
```

assigns the escape sequence for the **newline** character to the `char` variable `nextline`.

> Note
>
> In ANSI C a character constant is a sequence of **one or more** characters enclosed in single quotes. Precisely how the value of such a constant is to be interpreted is left to the implementation.

Since character variables are accommodated in a byte, C regards chars as being a subrange of ints, (the subrange that fits inside a byte) and each ASCII character is for all purposes equivalent to the decimal integer value of the bit picture which defines it. Thus `'A'`, of which the ASCII representation is `01000001`, has the arithmetical value of 65 decimal. This is the decimal value of the sequence of bits 01000001, as you may easily verify. In other words, the memory representation of the `char` constant `'A'` is indistinguishable from that of the `int` constant, decimal 65. The upshot of this is that small `int` values may be stored in `char` variables, and `char` values may be stored in `int` variables! Character variables are therefore signed quantities restricted to the range $[-128, 127]$. However, it is a requirement of the language that the decimal equivalent of each of the printing characters be non-negative. We are assured then that in any C implementation in which a `char` is stored in an 8-bit byte, the corresponding `int` value will always be a non-negative quantity, whatever the value of the leftmost (sign) bit may be. Now, identical bit patterns within a byte may be treated as a negative quantity by one machine, as a positive by another. For ensuring portability of programs which store non-character data in `char` variables the `unsigned char` declaration is useful: it changes the range of `char`s to [0, 255].

> Note
>
> However, the ANSI extension `signed char` explicitly declares a `signed` character type to override, if need be, a possible default representation of `unsigned char`s.

One consequence of the fact that C does not distinguish between the internal representation of byte-sized `int`s and `char`s is that arithmetic operations which are allowed on `int`s are also allowed on `char`s! Thus in C, if you wish to, you may multiply one character value by another. I list the ASCII decimal equivalents of the character set of C. Here are some variables declared as `char`s, and defined as escape sequences:

```
char newline = '\n', single_quote ='\"';
```

Character constants can also be defined via their **octal ASCII codes**. The octal value of the character, which you may find from the table in Appendix I, is preceded by a backslash, and is enclosed in single quotes:

```
char terminal_bell = \07';/* 7=octal ASCII code for beep */
char backspace = \010';/* 10=octal code for backspace */
```

> Note
>
> For ANSI C compilers, character constants may be defined by hex digits instead of octals. Hex digits are preceded by x, unlike 0 in the case of octals. Thus is ANSI C:
> `char backspace ='\xA';`
> is an acceptable alternative declaration to
> `char backspace ='\010';`
> Any number of digits may be written, but the value stored is undefined if the resulting character value exceeds the limit of `char`.

On an ASCII machine both '`\b`' and '`\010`' are equivalent representations. Each will print the backspace character. But the latter form, the ASCII octal equivalent of '`\b`', will not work on an EBCDIC machine, typically an IBM mainframe, where the **collating sequence** of the characters (i.e., their gradation or numerical ordering) is different. In the interests of portability therefore it is preferable to write '`\b`' for the backspace character, rather than its octal code. Then your program will work as certifiably on an EBCDIC machine as it will on an ASCII.

Note that the character constant 'a' is not the same as the string "a". (We will learn later that a string is really an **array** of characters, a bunch of characters stored in consecutive memory locations, the last location containing the null character; so the string "a" really contains two chars, 'a' immediately followed by '\0'). It is important to realise that the null character is not the same as the decimal digit 0, the ASCII code of which is 00110000.

Just as %d in printf() or scanf() allows us to print and read ints, %c enables the input and output of singe characters which are the values of char variables. Let's look at Programs 10 and 11 below: c

```
/* Program 10; file name: unit2-prog10.c */
#include <stdio.h>
int main()
{
    char a = 'H', b = 'e', c = 'l', d = 'o', newline = '\n';
    printf("%c", a);
    printf("%c", b);
    printf("%c", c);
    printf("%c", c);
    printf("%c", d);
    printf("%c", newline);
    return (0);
}
```

The output of Program 10 is easily predictable (what is it?).

```
/* Program 11; file name:unit2-prog11.c */
#include <stdio.h>
int main()
{
    char val_1 = 'a', val_2 = 'z';
    int val_3;
    printf("The first character is %c \
and its decimal equivalent is %d.\n", val_1, val_1);/*Continued*/
    printf("The second character is %c \
and its decimal equivalent is %d.\n", val_2, val_2);/*Continued*/
    val_3 = val_1 * val_2;
    printf("Their product is %d\n", val_3);
    return (0);
}
```

Execute the program above to verify that the char variables behave like one byte ints in arithmetic operations.

The %c format conversion character in a printf() outputs escape sequences, as you saw in Program 10. Execute Program 12 if you want a little music:

```
/* Program 2.12;File name:unit2-prog12.c */
#include <stdio.h>
int main()
{
    char bell = '\007'; /* octal code of the terminal bell */
    char x = 'Y', y = 'E', z = 'S', exclam = '!';
    printf("Do you hear the bell ? %c%c%c%c%c%c%c",
bell, x, bell, y, bell, z, exclam);/*Continued*/
    return (0);
}
```

Program 13 assigns a character value to an int variable, an integer value to a char variable, and performs a computation involving these variables. Predict the output of the program, and verify your result by executing it.

```
/* Program 13; file name: unit2-prog13.c*/
#include <stdio.h>
```

```
int main()
{
   int alpha = 'A', beta;
   char gamma = 122;
   beta = gamma - alpha;
   printf("beta seen as an int is: %d\n", beta);
   printf("beta seen as a char is: %c\n", beta);
   return (0);
}
```

One character that is often required to be sensed in C programs is not strictly speaking a character at all: it's the EOF or End-Of-File character, and its occurrence indicates to a program that the end of terminal or file input has been reached. Because the EOF is not a character, being outside the range of `char`s, any program that's written to sense it must declare an `int` variable to store character values. As we see in Program 13, this is always possible to do. An `int` variable can accommodate all the characters assigned to it, and can also accommodate EOF. We'll see uses for EOF in later units.

The `putchar`() function (pronounced "put character"), which takes a character variable or constant as its sole argument, is often a more convenient alternative for screen output than is the `printf`(). When this function is invoked (for which purpose you may need to `#include<stdio.h>`), the character equivalent of its argument is output on the terminal, at the current position of the cursor:

```
putchar (char_var);
```

Suppose `char_var` has been assigned the value `'A'`. Then `'A'` will be displayed where the cursor was.

Reciprocally, `getchar`() (pronounced "get character") gets a single character from the keyboard, and can assign it to a `char` variable. (`stdio.h` may have to be `#include`ed before `getchar`() can be used.) It has no arguments, and is typically invoked in the following way:

```
char_var = getchar();
```

When such a statement is encountered, the execution of the program is stayed until a key is pressed. Then `char_var` is assigned the character value of the key that was pressed.

Program 14 below illustrates the usage of these functions. Your compiler may require the inclusion of `stdio.h` to invoke `getchar`() and `putchar`().

```
/* Program 14; file name:unit2-prog14.c */
#include<stdio.h>
int main()
{
    char key_pressed;
    printf("Type n a lowercase letter (a-z), press <CR>:");
    key_pressed = getchar();    /* get char from keyboard */
    printf("You pressed ");
    putchar(key_pressed - 32);
    /* put uppercase char on Terminal */
    putchar('\n');
    /* converts to uppercase beacause
    ASCII decimal equivalent is 32
    less than for the corresponding
    lower case character. */
    return (0);
}
```

Note

There are certain characters required in **C** programs which are not available on the keyboards of many non-ASCII computers. In **ANSI C** these characters can be simulated by **trigraph** sequences, which are sequences of three characters of the form `??x`. They're treated as special characters even if they are embedded inside character strings.

Trigraph	Substitutesfor
??=	#
??(	[
??)	]
??<	{
??>	}
??/	\
??'	^
??!	|
??-	~

Here are some exercises to give you some practice in working with `char`.

E8) In Program 13 insert the declaration:
`unsigned char delta = alpha-gamma;`
and print the values of `delta` as an `int`, an `unsigned int`, a `char` and as an `unsigned char` quantity.

E9) Write a program which gets a character via `getchar`(), and prints its ASCII decimal equivalent.

We close this section here. In the next section, we will take up another data type called float that is used for storing fractions and irrational numbers.

2.4 VARIABLES OF TYPE float

Integer and character data types are incapable of storing numbers with fractional parts. Depending on the precision required, **C** provides two variable types for computation with "floating point" numbers, i.e. numbers with a decimal (internally a binary) point. Such numbers are called `floats` because the binary point can only notionally be represented in the binary-digits expansion of the number, in which it is made to "float" to the appropriate "position" for optimal precision. (You can immediately see the difficulty of imagining a binary "point" within any particular bit of a floating point word, which can contain only a 0 or a 1!) Typically, some of the leftmost bits of a floating point number store its characteristic (the positive or negative power of two to which it is raised), and the remaining its mantissa, the digits which comprise the number. In base 10, for example, if 2.3 is written as 0.0023×10^3, the mantissa is 0.0023, and the characteristic (exponent) is 3.

Single precision floating point variables are declared by the `float` specification, for example:

```
 float bank_balance = 1.234567E8;
/* En means 10 raised to the power n*/}
```

The scientific notation `En`, where the lowercase form `en` is also acceptable, is optional; one may alternatively write:

```
float bank_balance = 123456700.0;
```

Floats are stored in four bytes and are accurate to about seven significant digits; on PCs their range extends over the interval [E-38, E37].

It must never be lost sight of that the floating point numbers held in a computer's memory are at best **approximations** to real numbers. There are two reasons for this shortcoming. First, the finite extent of the word size of any computer forces a truncation or round-off of the value to be stored; whether a storage location is two bytes wide, or four, or even eighth, the value stored therein can be precise only to so many binary digits.

Second, it is inherently impossible to represent with unlimited accuracy some fractional values as a finite series of digits preceded by a binary or decimal point. For example

1/7 = 0.142857142857142857... ad infinitum

As long as a finite number of digits is written after the decimal point, you will be unable to accurately represent the fraction 1/7. But rational fractions that expand into an infinite series of decimals aren't the only types of floating point numbers that are impossible to store accurately in a computer. Irrational numbers such as the square root of 2 have aperiodic (non-repeating) expansions-there's no way that you can predict the next digit, as you could in the expansion of 1/7 above, at any point in the series. Therefore it's inherently impossible to store such numbers infinitely accurately inside the machine. The number π, which is the ratio of the circumference of any circle to its diameter, is another number that you cannot represent as a finite sequence of digits. As you know it's not merely irrational, it's a transcendental number. (It cannot be the root of any algebraic equation with rational coefficients.) So, an element of imprecision may be introduced by the very nature of the numbers to be stored.

Third, in any computation with floating point numbers, errors of round-off or truncation are necessarily introduced. For, suppose you multiply two n-bit numbers; the result will in general be a 2n-bit number. If this number (2n) of bits is larger than the number of bits in the location which will hold the result, you will be forcing a large object into a small hole! Ergo, there'll be a problem! Some of it will just have to be chopped off. Therefore it is wisest to regard with a pinch of salt any number emitted by a computer as the result of computation, that has a long string of digits after the decimal point. It may not be quite as accurate as it seems.

The `%e`, `%f` and `%g` format conversion characters are used in the `scanf()` and `printf()` functions to read and print floating point numbers `%e` (or `%E`) is used with floating point numbers in exponent format, while `%g`(or `%G`) may be used with `floats` in either format, `%g` (or `%G`) in the `printf()` outputs a `float` variable either as a string of decimal numbers including a decimal point, or in exponent notation, whichever is shorter. An uppercase E or G prints an uppercase E in the output. We shall be discussing more about format control in a later Unit.) Program 15 below finds the average of five numbers input from the keyboard, and prints it:

```
/* Program 15; file name:unit2-prog15.c */
#include<stdio.h>
int main()
{
    float val_1, val_2, val_3, val_4,
    val_5, total = 0.0, avg;
    printf("\nEnter first number...");
    scanf("%f", &val_1);
    printf("\nEnter second number...");
    scanf("%f", &val_2);
    printf("\nEnter third number...");
    scanf("%f", &val_3);
    printf("\nEnter fourth number...");
```

```
    scanf("%f", &val_4);
    printf("\nEnter fifth number...");
    scanf("%f", &val_5);
    total = val_1 + val_2 + val_3 + val_4 + val_5;
    avg = total / 5;
    printf("\nThe average of the numbers \
you entered is: %f\n", avg);/*Continued*/
    return (0);
}
```

Here's a sample conversation with the program:

```
 Enter first number...  32.4
Enter second number...  56.7
 Enter third number...  78.3
Enter fourth number...  67.8
Enter fifth number...  -93.9

The average of the numbers you entered is:  28.260000
```

You can check your understanding of our discussion of floats by doing the following exercise.

E10) Write a program to compute simple interest: If a principal of P Rupees is deposited for a period of T years at a rate of R per cent, the simple interest I is $I = \frac{PRT}{100}$. Your program should prompt for `float`s P, R and T from the keyboard and output the interest I.

We end this secton here. In the next section, we discuss **double**, a data type for storing larger floating numbers.

2.5 VARIABLES OF TYPE double

Because the words of memory can store values which are precise only to a fixed number of figures, any calculation involving floating point numbers almost invariably introduces **round-off** errors. At the same time scientific computations often demand a far greater accuracy than that provided by **single precision** arithmetic, i.e., arithmetic with the four-byte float variables. Thus, where large scale scientific or engineering computations are involved, the double declaration becomes the natural choice for program variables. The double specification allows the storage of `double precision` floating point numbers (in eight consecutive bytes) which are held correct to 15 figures, and have a much greater range of definition than floats, [E-308, E307]. Older compilers may also allow the long float specification instead of double, but its use is not recommended.

```
double lightspeed = 2.997925E10, pi= 3.1415928;
```

The `%lf` (or `%le` or `%lg`) specification in a `scanf`() is required for the input of `double` variables, which are however output via `%e` (or `%E`), `%f` or `%g` like their single precision counterparts. Program 16 which computes the volume of a cone, whose base radius and height are read off from the keyboard, illustrates this:

```
/* Program 16; file name:unit2-prog16.c */
#include <stdio.h>
#define PI 3.1415928
int main()
{
    double base_radius, height, cone_volume;
    printf("This program computes the volume of a cone\n");
    printf("of which the radius and height are entered\n");
```

```
    printf("from the keyboard.\n\nEnter radius of cone base:");
    scanf("%lf", &base_radius);
    printf("\nEnter height of cone:");
    scanf("%lf", &height);
    printf("\nVolume of cone of base radius R and height H \
is (1/3)*PI*R*R*H\n");/*Continued*/
    cone_volume = PI * base_radius * base_radius * height / 3;
    printf("\nVolume of cone is: %f\n", cone_volume);
    return (0);
}
```

One new feature of Program 16 is to be found in its second line.

```
#define PI 3.1415928
```

The **#defines—also** called MACRO definitions—are a convenient way of declaring constants in a C program. Like the **#include**s, the **#define**s are preprocessor control lines. #define**s** are processed at an early stage of the compilation. They cause the substitution of the named identifier by the associated token string throughout the text of the part of the program which follows the #define. Exception: not if the identifier is embedded inside a comment, a quoted string, or a char constant. Typically, a #define is of the form:

```
#define IDENTIFIER
identifier_will_be_replaced_by_this_stuff
```

In Program 16 above wherever PI occurs in the program, it is replaced by 3.1415928. (There is a longstanding tradition in C for writing names for MACRO constants in uppercases characters, and we will follow this practice.)

Like the #include, #define**s** are also not terminated by a semicolon; if they were, the semicolon would become part of the replacement string, and this could cause syntactical errors. For example consider the following program:

```
/* Program 17; file name:unit2-prog17.c */
#define PI 3.1415928;           /* Error! */
int main()
{       /* finds volume of cone, height H, base radius R */
    double H = 1.23, R = 2.34, cone_volume;
    cone_volume = PI * R * R * H / 3;
    return (0);
}
```

When the replacement for PI is made, the assignment for cone_volume takes the form:

```
cone_volume = 3.1415928;*2.34*2.34*1.23/3;
```

which cannot be compiled. For precisely the same reason the assignment operator = cannot occur in a MACRO definition.

A #define**d** quantity is **not** a variable and its value cannot be modified by an assignment. Though the MACRO definition for PI has been placed before main() in Program 17, this is not a requirement: it may occur anywhere in the program before it is referenced. One great convenience afforded by MACRO definitions is that if the token string representing the value of a #define**d** quantity has to be changed, it need be changed only once, in the MACRO itself. When the program is compiled, the changed value will be recorded in every occurrence of the quantity following the #defin**i**tion. So if you wish to make a computation in which the value of PI must be accurate to 16 places of decimals, just change it where it's #define**d**.

Note

ANSI C has another floating point type called long double which has at least as large a number of significant figures, and as large a range of allowable exponents as double. The system file `float.h` contains constants pertaining to floating point arithmetic. Some of these for gcc on 32 bit Linux are:

DBL_DIG	decimal digits of precision	15
FLT_DIG	decimal digits of precision	6
LDBL_DIG	decimal digits of precision	18
DBL_MANT_DIG	bits to hold the mantissa	53
FLT_MANT_DIG	bits to hold the mantissa	24
LDBL_MANT_DIG	bits to hold the mantissa	64
DBL_MAX_10_EXP	maximum exponent values	308
FLT_MAX_10_EXP	maximum exponent values	38
LDBL_MAX_10_EXP	maximum exponent values	4932
DBL_MIN_10_EXP	minimum exponent values	-307
FLT_MIN_10_EXP	minimum exponent values	-37
LDBL_MIN_10_EXP	minimum exponent values	-4931

You can always print these values for your machine by writing a small C program similar to the one we wrote for integers.

Note

C compilers convert all single precision floating-point constants to double precision wherever they occur in a computation, so it's as well to declare all floating point constants in a program as `double`. In ANSI C the suffix `f` or `F` after a floating point value forces it to be treated as a single precision constant; the suffix `l` or `L` treats as a `long double` constant. Either the integer part or the fraction part may be absent from the definition of a floating constant value; not both of the decimal point and the e and its exponent may be missing.

Here is a version of Simpson rule that uses `#define` to define the integrand.

```
/*Numerical integration on the interval [0,1] using Simpson's
Rule. It uses 6 sub intervals. File name:unit2-simpson.c*/
#include<stdio.h>
#include<math.h>
#define f(x) (1/(1+(x)*(x)))/* You can change this.*/
/* Note that the previous line does not contain a =
 between f(x) and 1/(1+(x)*(x))*/
int main()
{
    float h = 1 / 4.0;
    printf("The value of the integral is \
%f\n", (h / 3.0) * (f(0) + 4 * (f(h) +/*Continued*/
f(3 * h)) + 2 * f(2 * h) + f(4 * h)));/*Continued*/
    return (0);
}
```

Note

ANSI C provides the **const** declaration for items whose values should not change in a program:

const int `dozen = 12;`

The keyword `const` lets the programmer specify the type explicitly in contrast to the `#define`, where the type is deduced from the definition. In addition, ANSI C has a modifier volatile, to explicitly indicate variables whose values may change, and which must be accessed for a possibly changed value whenever they are referenced.

E11) Write a C program to compute the volume of a sphere of radius 10. The volume is given by the formula:

$$V = \frac{4}{3}\pi R^3$$

where R is the radius of the sphere.

In the next section, we will discuss some other data types called enumerated types.

2.6 ENUMERATED TYPES

In addition to these four types of program variables, C allows additional user-defined variable types, called `enum` (from enumerated) types. Consider the following declaration.

```
enum grades
{
F,D,C_MINUS, C, C_PLUS, B_MINUS, B, B_PLUS,
A_MINUS, A, A_PLUS
}result;
```

This declaration makes the variable `result` an enumerated type, namely `grades`, which is called the **tag** of the type. The tag is a reference to the enumerated type, and may be used to declare other variables of type `enum grades:`

```
enum grades final_result;
```

The variables `result` and `final_result` can only be assigned on of the values `F, D,..., A_PLUS`, and no others. These eleven enumerators have the consecutive integer values 0 (for F) through 10 (for A_PLUS); so that if you assign the value `A_PLUS` to `result` and later examine it, it will be found to be 10 (and not the string "A_PLUS". Though the C compiler stores enumerated values as integer constants, `enum` variables are a distinct type, and they should not be thought of as `int`s. [In ANSI C the enumerators are `int` constants.]

In an `enum` list any enumerator may be specified an integer value different from the constant associated with it by default. The enumerator to its right gets a value 1 greater, and further down the list, each enumerator becomes 1 more than the preceding.

```
enum flavours
{
sweet, sour, salty = 6, pungent, hot, bitter
} pickles;
```

In the declaration above `sweet` and `sour` have the values 0 and 1 respectively, while `pungent, hot` and `bitter` are 7, 8 and 9 in that order.

2.7 THE typedef STATEMENT

The typedef statement is used when one wants to refer to a variable type by an alternative name, or alias. This often makes a great convenience for the programmer: suppose you are writing a program to keep track of all the cutlery—spoons, forks, knives and serving ladles_in a restaurant. Then, the statement:

```
typedef int cutlery;
```

will enable you to declare:

```
cutlery spoons, forks, knives, serving_ladles;
```

which may be more meaningful than:

```
int spoons, forks, knives, serving_ladles;
```

`cutlery` becomes an alias for int. typedef does no more than rename an existing type.

E12) Name the types of C variables you would choose to represent the following quantities—`char, int, float, double` or `enum:`

1) The velocity of sound in air, approximately 750 miles/hour.
2) The velocity of light—in vacuum, 2.997925E10 cm/sec.
3) The number of seconds in 400 years.
4) The punctuation symbols of the English alphabet.
5) The months of the year.
6) The population of the city of Delhi, approximately 9 million.
7) The population of planet Earth, approximately 5.4 million.
8) The value of Avogadro's number, 6.0248E23.
9) The value of Planck's constant, 6.61E-27.
10) 6,594,126,820,000,000,000,000, which is the approximate value of the mass. of the earth, in tons.
11) The colours of the rainbow.
12) The II class train fare between any two cities in India.
13) The seat capacity of a Boeing 747.
14) The vowels of the English alphabet.
15) The days of the week.
16) The savings bank balance of Rs. 12,345.67.
17) The savings bank balance of Rs. 12,34,56,89.10.
18) The square-root of 6 correct to 6 significant figures.
19) The cube-root of 15 correct to 15 significant figures.

So far we have merrily given names to variables in C programs without bothering about their validity. However, there are some rules governing variable names. We will discuss these rules in the next section.

2.8 IDENTIFIERS

We have already seen several examples of C identifiers, or names for program variables (more precisely, the name of storage locations): `newline, octal_num, apples, volt`, etc. Identifiers for variables and constants, as well as for functions, are sequences of characters chosen from the set {A-Z, a-z, 0-9,_}, of which the first character must not be a digit. C is a case sensitive language, so that `ALFA` and `ALFa` are **different** identifiers, as are `main`() and `Main`(). The underscore character (_) should not be used as the first character of a variable name because several compiler defined identifiers in the standard C library have the underscore for the beginning character, and inadvertently duplicated names can cause "definition conflicts". Identifiers may be any reasonable length; generally 8-10 characters should suffice, though certain compilers may allow for very much longer names (of up to 63 characters).

Note

ANSI C: According to ANSI C standards, **at least** first 31 characters of internal variables are significant.(All the variables that we have seen so far are internal variables only.) In other words, two identifiers must be regarded as different if at least one of the first 31 of their characters are different. We will see what are external variables in the next block. For external variables, **at least** 6 characters are significant.

C has a list of **keywords** e.g. `int, continue`, etc. which cannot be used in any context other than that predefined in the language. (This implies, for example, that you can't have a program variable named `int.` A list of keywords is appended below; take care that you do not choose identifiers from this list. Your programs will not compile. Indeed one should consistently follow the practice of choosing names for variables which indicate the roles of those variables in the program. For example, the identifier `saving_balance` in a program that processes savings-bank balances is clearly a better choice for representing the variables than is `asdf`.

C Keywords

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

E13) The output of the program below on an ASCII machine was the alphabetical character e. What is the ASCII decimal equivalent of d?

```
/* Program 2.18; File name:unit2-prog18.c*/
#include <stdio.h>
int main()
{
    char a, b, c = 'd';
    b = c / 10;
    a = b * b + 1;
    putchar(a);
    return (0);
}
```

E14) State the output of programs 19 and 20, and verify your results on a computer:

```
/* Program 2.19; File name:unit2-prog19.c */
#include <stdio.h>
int main()
{
    int alpha = 077, beta = 0xabc, gamma = 123, q;
    q = alpha + beta - gamma;
    printf("%d\n", q);
    q = beta / alpha;
    printf("%d\n", q);
    q = beta % gamma;
    printf("%d\n", q);
    q = beta / (alpha + gamma);
    printf("%d\n", q);
    return (0);
}
```

```
/* Program 2.20;File name unit2-prog20.c */
#include <stdio.h>
int main()
{
    char c = 72;
    putchar(c);
    c = c + 29;
    putchar(c);
    c = c + 7;
    putchar(c);
    putchar(c);
    c = c + 3;
    putchar(c);
    c = c - 67;
    putchar(c);
    c = c - 12;
    putchar(c);
    c = c + 87;
    putchar(c);
    c = c - 8;
    putchar(c);
    c = c + 3;
    putchar(c);
    c = c - 6;
    putchar(c);
    c = c - 8;
    putchar(c);
    c = c - 67;
    putchar(c);
    putchar('\n');
    return (0);
}
```

E15) Write C programs to verify whether:

$$36^2 + 37^2 + 38^2 + 39^2 + 40^2 = 41^2 + 42^2 + 43^2 + 44^2$$
$$23^3 + 24^3 + 25^3 = 204^2$$
$$5^8 + 12^8 + 13^8 = 59^4 + 120^4 + 179^4$$

E16) The Fibonacci numbers F_1, F_2, F_3,..., F_n are defined by the relations:
$F_1 = 1$
$F2 = 1$
$F_n = F_{n-1} + F_{n-2}, n > 2$
F_3 and successive numbers of the series are obtained by adding the preceding two numbers. For large n the ratio of two consecutive Fibonacci numbers is approximately 0.618033989. Given that F(100) is

354, 224, 848, 179, 261, 915, 075(21 digits)

Write a C program to find approximations to F(99) and F(101).

E17) The Lucas numbers are defined by the same recurrence relation as the Fibonacci's, where L_1 is 1 but L_2 is 3. Write a C program to print the first 10 Lucas numbers.

E18) The first large number to be factorised by a mechanical machine was one of 19 digits:

1, 537, 228, 672, 093, 301, 419.

The two prime factors found (which were known beforehand) were 529,510,939 and 2,903,110,321. Writing about that event later in Scripta Mathematica 1 (1933) [quoted by Malcolm E Lines in Think of a Number, Pub. Adam Hilger, 1990] Professor D. N. Lehmer (who with his son had invented the machine) remarked,

"It would have surprised you to see the excitement in the group of professors and their wives, as they gathered around a table in the laboratory to discuss, over coffee, the mysterious and uncanny powers of this curious machine. It was agreed that it would be unsportsmanlike to use it on small numbers such as could be handled by factor tables and the like, but to reserve it for numbers which lurk as it were, in other galaxies than ours, outside the range of ordinary telescopes."

Write a C program to determine, to the best extent you can, whether the two factors found are indeed correct.

We close the Unit here. In the next section, we will summarise the Unit.

2.9 SUMMARY

The fundamental data types of C are declared through the seven keywords: `int, long, short, unsigned, char, float` and `double`. (The keywords `signed` and `long double` are ANSI C extensions.)

1. Integers: ranges are in general machine dependent, but in any implementation a `short` will be at most as long as an `int`, and an `int` will be no longer than `long`. `Unsigned` integers have zero or positive values only.
2. Characters: `char` variables are used to represent byte-sized integers and textual symbols.
3. Floating point numbers: may be single or double precision numbers with a decimal point. Unless specified to the contrary (in ANSI C), single precision `float`s are automatically converted to `double` in a computation. `double` variables allow a larger number of significant figures, and a larger range of exponents than `float`s. `getchar`() gets a keystroke from the keyboard; `putchar`() deposits its character argument on the monitor.
4. `printf`() uses the following format conversion characters to print variables:

d	decimal integers
u	unsigned integers
o	octal integers
l	long
x	hex integers, lowercase
X	hex integers, uppercase
f	floating point numbers
e	floating point numbers in exponential format, lowercase e
E	floating point numbers in exponential format, uppercase E
g	floating point numbers in the shorter of f or e format
G	floating point numbers in the shorter of f or E format c single characters

   The modifier # with `x` or `X` causes a leading `0x` or `0X` to appear in the output of hexadecimal values; `scanf`() uses `%lf` to read `doubles`.

2.10 SOLUTIONS/ANSWERS

E1) Here x/y gives the quotient of x when divided by y. When x = 60, y = 15, x/y is 4; when x = 70, y = 15 x/y is 4; when x = 75, y = 15, x/y is 5 etc.

E2) Here x % y gives remainder when x is divided by y if y < y. It gives y if x < y and 0 if x = y.

E3) Here is the program:

```
/* Answer to Exercise 2.3; File name:unit2-ex3ans.c*/
#include<stdio.h>
int main()
{
    unsigned cheque = 54321;
    int seconds, minutes, hours, days;
    long time = 1234567890L;    /*seconds */
    printf("I\'ve waited a long time (%ld seconds)\n", time);
    printf("for my cheque (for Rs.%u/-), and now\n", cheque);
    printf("I find it\'s unsigned!\n");
    printf("That's a wait of:\n");
    seconds = time % 60;
    time = time / 60;
    minutes = time % 60;
    time = time / 60;
    hours = time % 24;
    days = time / 24;
    printf("%d days\n", days);
    printf("%d hours\n", hours);
    printf("%d minutes\n", minutes);
    printf("%d seconds\n", seconds);
    return (0);
}
```

E4) Here is the example program:

```
/*Answer to exercise 4; File name: unit2-ex4ans.c*/
#include<stdio.h>
int main()
{/* p is in octal notation, q is in hex. */
    int p = 012345, q = 0x1234;
    long octal_num = 0123456701L, hex_num = 0x7BCDEF89L;
    printf("Value of p is %#o\n", p);
    printf("Value of q is %#lx\n", q);
    printf("Value of octal_num is %#lo\n", octal_num);
    printf("Value of hex_num is %#lX\n", hex_num);
    return (0);
}
```

E5) The following is the output on a 32-bit Linux machine when the program is compiled with gcc:

```
x = 5
x = 25
x = 625
x = 390625
x = -2030932031
```

E6)

```
/*Answer to exercise 6; Filename: unit2-printlimitsfull.c*/
#include<stdio.h>
#include<limits.h>
int main()
{
    printf("Number of bits in a char is: %d\n", CHAR_BIT);
    printf("Maximum value of char is %d: \n", CHAR_MAX);
    printf("Minimum value of char is %d: \n", CHAR_MIN);
    printf("Maximum value of int is %d: \n", INT_MAX);
    printf("Minimum value of int is %d: \n", INT_MIN);
    printf("Maximum value of long is %ld: \n", LONG_MAX);
    printf("Minimum value of long is %ld: \n", LONG_MIN);
    printf("Maximum value of signed \
```

```
char is %d: \n", SCHAR_MAX);/*Continued*/
    printf("Minimum value of signed is %d: \n", SCHAR_MIN);
    printf("Maximum value of short is %d: \n", SHRT_MAX);
    printf("Minimum value of short is %d: \n", SHRT_MIN);
    printf("Maximum value of unsigned \
char is %d: \n", UCHAR_MAX);/*Continued*/
    printf("Maximum value of unsigned \
int is %u: \n", UINT_MAX);/*Continued*/
    printf("Maximum value of unsigned \
long is %lu: \n", ULONG_MAX);/*Continued*/
    return (0);
}
```

E7)
```
/*Program that checks for overflow.
File:unit2-ex7ans.c*/
#include<stdio.h>
#include<limits.h>
int main()
{
    unsigned long x, y;
    printf("Enter the value of x...\n");
    scanf("%lu", &x);
    printf("Enter the value of y...\n");
    scanf("%lu", &y);
    if (x > ULONG_MAX - y)
        printf("The sum of x and y exceeds\
the limit of ULONG_MAX\n");/*Continued*/
    else
        printf("\nThe sum of x and y is %lu\n", x + y);
    return 0;
}
```

E8)
```
/* Program ans-ex7; file name: prog13-ex7.c*/
#include <stdio.h>
int main()
{
    int alpha = 'A', beta;
    char gamma = 122;
    unsigned char delta = alpha - gamma;
    beta = gamma - alpha;
    printf("beta seen as an int is: %d\n", beta);
    printf("beta seen as a char is: %c\n", beta);
    printf("delta as an int %d\n", delta);
    printf("delta as an unsigned int %u\n", delta);
    printf("delta as a char is %c\n", delta);
    return (0);
}
```

The output is

beta seen as an int is: 57

beta seen as a char is: 9

delta as an int 199

delta as an unsigned int 199

delta as a char is

E9)
```
/*Answer to exercise 9; File name unit2-ans-ex9.c*/
#include<stdio.h>
int main()
{
    int i;
    printf("Enter the character...\n");
```

```
    i = getchar();
    printf("The decimal equivalent is %d.", i);
    return 0;
}
```

E10)
```
/*Program to calculate simple interest.
File name: unit2-ansex10.c*/
#include<stdio.h>
int main()
{
    double principal, rate_of_interest, period,interest;
    printf("This program calculates simple interest\n");
    printf("Enter the principal\n");
    scanf("%lf", &principal);
    printf("Enter the rate of interest:\n");
    scanf("%lf", &rate_of_interest);
    printf("Enter the period in months:\n");
    scanf("%lf", &period);
    interest=(principal * rate_of_interest * period) / 100;
    printf("The simple interest is \n");
    printf("%lf", interest);
    return (0);
}
```

E11)
```
/*Program to calculate the volume of a sphere
File name: unit2-ansex11.c*/
#include<stdio.h>
#define PI 3.1415928
int main()
{
    double radius;
    printf("Enter the radius of the sphere:\n");
    scanf("%lf",&radius);
    printf("\n The volume of the sphere is:\n");
    printf("%lf",4*PI*radius*radius*radius/3);
    return (0);
}
```

E12)

1)	float	11)	enum
2)	float	12)	float
3)	It is an integer type, but none of the integer types is big enough. So, we have to use float.	13)	short int
4)	char	14)	char
5)	enum	15)	enum
6)	int	16)	float
7)	float	17)	float
8)	float	18)	float
9)	float	19)	double
10)	double		

E13) 100

E14) The output from program 19 is

2688

43

42

14

The output from program 20 is

E15)
```
/*Answer to exercise 14; File name unit2-ans-ex14.c*/
#include<stdio.h>
long sq(int x);
long cu(int x);
main()
{
    long ans, ans1;
    ans = sq(36) + sq(37) + sq(38) + sq(39) + sq(40);
    ans1 = sq(41) + sq(42) + sq(43)+sq(44);
    printf("%ld", ans - ans1);
    ans = cu(23) + cu(24) + cu(25);
    ans1 = sq(204);
    printf("\n%ld", ans - ans1);
    ans = cu(sq(5)) * sq(5) + cu(sq(12)) * sq(12)
+ cu(sq(13)) * sq(13);/*Continued*/
    ans1 = sq(sq(59)) + sq(sq(120)) + sq(sq(179));
    printf("\n%ld", ans - ans1);
    return 0;
}
long sq(int x)
{
    return x * x;
}
long cu(int x)
{
    return x * x * x;
}
```

E16)
```
/*Answer to exercise 15.*/
/*File name: unit2-ans-ex15.c*/
#include<stdio.h>
int main()
{
    long double f99, f100 = 354224848179261915075.0L, f101;
    long double ratio = 0.618033989L;
    printf("\nf100=%Lf\n", f100);
    f99 = f100 * ratio;
    f101 = f100 / ratio;
    printf("f99 = %Lf\n f100 = %Lf\n f101 = %Lf\n",
f99, f100, f101);/*Continued*/
    printf("f101-f100-f99=%Lf", f101 - f99 - f100);
    return 0;
}
```

Note the %Lf format modifier for long double. %Le is also allowed for printing long double in exponential notation.

E17)
```
/*Answer to exercise 18; File name: unit2-ansex18.c*/
#include<stdio.h>
int main()
{
    long double prime_1 = 529510939.0L;
    long double prime_2 = 2903110321.0L, product;
    product = prime_1 * prime_2;
    printf("The product of the primes is %Lf", product);
    return 0;
}
```

Does the program give the correct answer?

ASCII Table

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	‘	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	’	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	∼	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

Note that the characters from 0–31 and character 127 are non-printing characters and character 32 is the space.

UNIT 3 OPERATORS AND EXPRESSIONS IN C

Structure **Page No.**

3.1 INTRODUCTION

This Unit introduces some basic concepts of the C language. If we have an expression like `2+7%3`, will the C program first add 2 and 7 and then find the reminder on division by 3 or will it add the reminder on division of 7 by 3 to 2? In Sec. 3.2, we will discuss rules that govern the evaluation in such situations. In Sec. 3.3, we discuss the concept of C expressions. In Sec. 3.4, we discuss **lvalues** and **rvalues**. These concepts are important pre-requisites for understanding arrays and pointers. When variables of two different types like **float** and **long int** occur in a computation, what will be the type of the result? We discuss such questions in Sec. 3.5. In Sec. 3.6, as we promised in the previous unit, we will discuss how to control the output and input in `printf()` and `scanf()` functions, respectively.

Objectives

After studying this unit, you should be able to

- write and evaluate complex C expressions, built with the arithmetic operators of C;
- explain the order of precedence among operators, and direction in which each associates;
- explain the concept of lvalue and rvalue of a variable;
- explain how these properties help decide the sequence in which the various parts of a C expression are evaluated; and
- explain how to use format control statements in `printf()` and `scanf()` to control input and output in C programs.

3.2 ELEMENTARY ARITHMETIC OPERATIONS AND OPERATORS

Let us start our discussion of arithmetic operators by the discussing a simple statement. Let us look at the statement. 3 times 4 is 12. The "times" or multiplication operation is represented in C, as it is in most computer languages, by the asterisk, *. In other words, the **operator** for **multiplication** is the *. It's a **binary operator** because it takes **two operands**, the multiplier and the multiplicand, namely 4 and 3 respectively in the present case. Now consider the C statement:

```
x=3*4;
```

There are **two** operators at work here. This may seem a little strange to you: the only visible operator, you might think, is the operator for multiplication, the asterisk. Not quite true. The second operator in the statement above is the **assignment operator**, the =. It assigns the value of the quantity on its right hand side to the variable named on its left, overwriting its previous value. The point to note is that **the assignment of a value to a variable is an operation**, the execution of a machine instruction, just as much as multiplication is an operation. The assignment operation causes the CPU to seek out the RAM location of the variable, and to deposit therein the value assigned to the variable.

The Random Access Memory in a computer may be thought of as a linear collection of words, each word being a basic unit of storage. Every word of memory has a unique "address", which is a positive integer that helps identify the word. The addresses of successive words increase uniformly from the lowest to the highest value. This is not unlike the way houses along a road have numbers by means of which they are located

Once the CPU is given the address of a word, selected at random, it can rapidly gain access to its contents, in a fixed small amount of time (of the order of a hundred millionth of a second), no matter what random value (between the lowest and the highest) the address of the word may have. Hence the name: Random Access Memory.

After a program has been compiled, each variable is assigned to a word of RAM. This association between a program variable and its memory address remains unchanged throughout the execution of the program. When you refer to a variable by its identifier, the computer translates the reference to the address of the word in which the variable is stored. The contents of a word may change as the program executes, just as much as the residents of a house may change with the passage of time. But for the duration of time that a family lives in a house, the association of the family's name with the address of the house is fixed.

Similarly, for as long as a program module executes, the association of a variable with its storage location remain fixed.

Now you can see why statements such as:

```
n = n + 1;
```

make sense to a computer. To a person unfamiliar with programming the reflex reaction on encountering such a statement is, "How can `n` possibly equal `n + 1`?" The fact is that in such a statement one is not asserting the equality of `n` to `n + 1`. One is actually instructing the CPU: "Replace the value in the storage location marked `n`, by whatever it was plus 1." To emphasis the difference between the act of assignment of a value to a variable, commonly done in computer programs, and the assertion of equality of the left and right hand side quantities in an algebraic equation, commonly done in high school problems, Pascal uses the compound symbol: = for assignment. In C the operator for assignment is the = symbol, and the statement:

```
n = n + 1;
```

has the meaning:

"Make the new contents of `n` its old contents + 1"

C operators have two properties with which you must become familiar: these are **priority** and **associativity**. When there is more than one operator occurring in an expression, it is the relative priorities of the operators with respect to each other that will determine the order in which the expression will be evaluated. The associativity defines the direction, left-to-right or right-to-left, in which the operator acts upon its operands. You may have noticed that we quietly slipped in a new word, "expression", without defining it first: here's the definition: an **expression** is a syntactically valid combination of operators and operators, that computes to a value. The number 7, or any

other number by itself, is an expression with no operands. It's a valid C expression. It's value is the value of the number, in the present case, 7. 3 + 8 is another valid expression, in which the operands 3 and 8 are syntactically connected by +, the operator for addition. Though we hate to be so obvious, it must be stated that this last expression computes to the value 11.

The assignment operator assigns the quantity on its right to the variable named on its left. In other words, **it groups, or associates from right to left**. In contrast, **the multiplication operator groups from left to right**, as indeed do most C operators. Basically, this means that if C sees a statement like:

```
w = x * y * z;
```

it will first compute `x*y`, (grouping from left to right), and will then multiply this value by z. Then, and only then, will it assign the value of the product `x*y*z` or `w`.

It is important to realise that the action of assignment is performed **after** the computations have been done. This is the natural order of arithmetic, and this, clearly, is what one should expect—first compute a value, then assign it to the variable on the left. Here then is the reason for the built in priority of C operators, and for the direction of their association.

It makes sense for the priority of the assignment operator to be lower than the priorities of an the arithmetic operators, and for it to group from right to left. Naturally it is very important for you to become adept at the precedence and grouping properties of all of C's operators. But if you are not sure of the order in which operators will be evaluated in a computation, you may use the **parentheses operator**, the (), to override default priorities. (Yes, even the parentheses is an operator in C!) The parentheses operator has a priority higher than any binary operator, such as that for multiplication; it groups from left to right. Thus in the statements:

```
w = x * (y * z);
```

the product `y * z` will be computed first; the value obtained will then be multiplied by `x`; lastly the assignment of the result will be made to `w`. Had the parentheses been absent, the order of the computation would have been:

1) The multiplication of `x` by `y`, with the result stored as an intermediate quantity.
2) The multiplication of this quantity by `z`.
3) The assignment of the result to `w`.

Generally we will state the direction of association and the relative priority of each operator of C with respect to the others when we introduce it. But for convenience a table (in order of decreasing operator priority) stating their direction of grouping is given in Table. 1 on the following page. [Not all of these operators may be available in non-ANSI implementations.] The parenthesis is an example of a **primary** operator; C has in addition three other primary operators: the **array operator** (`[]`), and the **dot** (`.`) and **arrow**`->` operators which we will encounter in later Units. All these operators have the same priority, higher than that of any other operator. They all group from left to right.

Aside from the primary operators, C operators are arranged in priority categories depending on the number of their operands. Thus a **unary** operator has but a single operand, and a higher priority than any **binary** operator, which has two operands. Binary operators have a higher priority than the **ternary** operator, with three operands. The **comma** operator may have any number of operands, and has the lowest priority of all C operators. Table. 1 on the next page reflects this rule.

One readily available example of a unary operator is the **operator for negation**, the `-`. It changes the sign of the quantity stated on its right. Since the unary operators have a higher priority than the assignment operator, in the statement:

Table 1: Precedence and Associativity of Operators.

Operators	Associativity
(), [], ->,.,	L to R
!, ~, ++, -,+,-, *derefencing operator, &, (type cast), sizeof, (all unary)	R to L
*,/,%,	L to R
+ , -	L to R
<<,>>	L to R
<, <=, >, >=	L to R
==,!=	L to R
&	L to R
^	L to R
|	L to R
&&	L to R
||	L to R
? : (Ternary if-then-else operator)	L to R
=, +=, -=, *=, /=, %=, &=, ^=, |=, «=, »=	R to L
, (comma operator)	L to R

```
x = -3;
```

the 3 is **first** negated, and only **then** is this value assigned to x. The negation operator has a priority just below that of the parentheses operator; it groups from right to left. (Right to left association is a property the operator for negation shares in common with all unary operators.) In the following statement:

```
x = -(3*4);
```

the presence of the parentheses ensures that the expression 3*4 is evaluated first. It is then negated. Finally x is assigned the value -12.

A question that you might have is: Does C have a unary plus operator, +? In other words can one make an assignment of the form a = + 5? Not in compilers conforming to the **K** & **R** standard, though ANSI C does provide a unary plus operator. See Table. 1.

As we learnt in the last unit, C provides operators for other elementary arithmetic operations, such as addition, subtraction, division and residue-modulo (the operation that yields the remainder after division of any integer by another). They are respectively +, -, / and % (Refer to Program 2.3 and 2.4). Here we quickly recapitulate their properties. Each of these operators requires two operands. (The binary operator for subtraction must be distinguished from the unary operator for negation.)

If int variables a, b, c and d are given the values 3, 4, 5 and 6 respectively, then:

```
a+b  is 7,   /*  +,  the operator for addition */
b-c  is -1,  /*  -,  the operator for subtraction */
a*b  is 12,  /*  *,  the operator for multiplication */
c/b  is 1,   /*  /,  the operator for division: the remainder is discarded
                     when one int is divided by another. */
a%d  is 3,   /*  %   the residue modulo operator; it yields the remainder
                     after division of a by d */
```

Important: In the division of one integer by another the remainder is discarded. **Thus 7/3 is 2, and 9/11 is 0.**

The multiplication, division and residue-modulo operators have each the same priority. The addition and subtraction operators also have equal priority, but this is lower than that of the former three operators. *, / and %. All these operators group from left to right.

In a C program, is the value of 3/5+2/5 the same as (3+2)/5? Is 3*(7/5) the same as 3*7/5? Let us look at some examples to understand how the priorities work in the case of arithmetic operators.

Example 1: In the examples below let `x` be an `int` variable:

i) `x = 2 * 3 + 4 * 5;`

The products `2*3` and `4*5` are evaluated first; the sum `6 + 20` is computed next; finally the assignment of 26 is made to `x`.

ii) `x = 2 * (3 + 4) * 5;`

The parentheses guarantee that `3 + 4` will be evaluated first. Since multiplication groups from left to right, the intermediate result 7 will be multiplied by 2, and then by 5, and the assignment of 70 will finally be made to `x`.

iii) `x = 7 * 6 % 15 / 9;`

Each of the operators above has equal priority; each groups from left to right. Therefore, the multiplication 7 * 6 (= 42) is done first, then the residue modulo with respect to 15 (42 % 15 = 12), and finally the division (of 12) by 9. Since the division of one integer by another yields the integer part of the quotient, and truncates the remainder, 12 / 9 gets the value 1. `x` is therefore asigned the value 1.

iv) `x = 7 * (6 % 15)/ 9;`

The parentheses ensure than 6 % 15 is evaluated first. The remainder when 6 is divided by 15 is 6. In the second step this result is multiplied into 7, yielding 42. Integer division of 42 by 9 gives 4 as the quotient, which is the value assigned to `x`.

v) `x = 7 * 6 % (15 / 9);`

15 / 9 is performed first, yielding 1; the next computation in order is 7 * 6 % 1, i.e. the remainder on division of 42 by 1, which is 0. `x` gets the value 0.

vi) `x = 7 * ((6 % 15)/ 9);`

The innermost parentheses are evaluated first: 6 % 15 is 6. The outer parentheses are evaluated next, 6/9 is 0. `x` gets the value 7 * 0 = 0.

* * *

Try some exercises now to check your understanding of priorities of operators.

E1) Verify by creating and executing a C program that for `int` variables `a` and `b`, `a % b` equals `a - (a / b) * b`, regardless of whether `a` or `b` is negative or positive.

E2) Find the value that is assigned to the variables `x, y` and `z` when the following program is executed:

```
/* Program3.1; File name:unit3-prog1.c*/
#include<stdio.h>
int main()
{
    int x, y, z;
    x = 2 + 3 - 4 + 5 - (6 - 7);
    y = 2 * 33 + 4 * (5 - 6);
    z = 2 * 3 * 4 / 15 % 13;
    x = 2 * 3 * 4 / (15 % 13);
```

```
    y = 2 * 3 * (4 / 15 % 13);
    z = 2 + 33 % 5 / 4;
    x = 2 + 33 % -5 / 4;
    y = 2 - 33 % -5 / -4;
    z = -2 * -3 / -4 % -5;
    x = 50 % (5 * (16 % 12 * (17 / 3)));
    y = -2 * -3 % -4 / -5 - 6 + -7;
    z = 8 / 4 / 2 * 2 * 4 * 8 % 13 % 7 % 3;
return (0);
}
```

By inserting appropriate calls to `printf()`, verify that the answers you obtained are correct.

E3) Give the output of the following program:

```
/* Program 3.2; File name:unit3-prog2.c */
#include <stdio.h>
int main()
{
    int x = 3, y = 5, z = 7, w;
    w = x % y + y % x - z % x - x % z;
    printf("%d\n", w);
    w = x / z + y / z + (z + y) / z;
    printf("%d\n", w);
    w = x / z * y / z + x * y / z;
    printf("%d\n", w);
    w = x % y % z + z % y % (y % x);
    printf("%d\n", w);
    w = z / y / y / x + z / y / (y / x);
    printf("%d\n", w);
    return (0);
}
```

We have reached the end of this section. In the next section, we will discuss the concept of expressions in C.

3.3 EXPRESSIONS

An `expression` in C consists of a syntactically valid combination of operators and operands, that computes to a value. An expression by itself is not a statement. Remember, a statement is terminated by a semicolon; an expression is not. Expressions may be thought of as the constituent elements of a statement, the "building blocks" from which statements may be constructed. The important thing to note is that **every C expression has a value**. The number 7, as we said a while ago, or any other number by itself, is also an expression, the value of the number being the value of the expression.

`3 * 4 % 5`

is an expression with value 2.

`x = 3 * 4`

is an example of an **assignment expression**. (Note the absence of the semicolons in the assignment above. The terminating semicolon would have converted the expression into a statement.) Like any other C expression, **an assignment expression also has a value. It's value is the value of the quantity on the right hand side of the assignment operator**. So, in the present instance the value of the expression `(x = 3 * 4)` is 12. Therefore it is, that in C, statements such as:

`z = (x = 3 * 4)/ 5;`

are meaningful. Here the parentheses ensure that x is assigned the value 12 first. **12 is also the value of the parenthetical expression (x = 3 * 4)**, from the property that every expression has a value. Thus, the entire expression reduces to:

```
z = 12/5
```

Next in order of evaluation is the integer division of 12 by 5, yielding 2. The leftmost assignment operator finally bestows the value 2 to z. x continues to have the value 12.

Consider now the expression:

```
x = y = z = 3
```

The assignment operator groups from right to left. Therefore the rightmost assignment:

```
z = 3
```

is made first. z get the value 3; this is also the value of the rightmost assignment expression, `z = 3`. In the next assignment towards the left the expression is:

```
y = z = 3.
```

since the sub-expression `z = 3` has the value 3, we are saying in effect:

```
y = (z = 3)
```

i.e.

```
y = 3
```

the assignment to y is again of the value 3. Equally,the entire expression:

```
y = z = 3
```

gets the value 3. In the final assignment towards the left x gets the value of this latter expression:

```
x = (y = (z = 3))
```

Value of each parenthetical expression is 3. Thus x is 3. `printf()` prints expressions just as easily as it prints variables. This is illustrated in the program in Listing 1 below:

```
/* Program 3.3; File name:unit3-prog3.c */
#include <stdio.h>
int main()
{
    int x, y, z;
    printf("x = %d\n", x = (y = 2 * (z = 12 * 13 / 14)) % 5);
    printf("x = %d\n", x = 3 * (y = x % (z = 2 * (x = 5))));
    printf("x = %d\n", x = 2 * (x = z % (y = 10 -
          (x = 3 * (z = 13 % (x = 3 * (y = 12 / (z = 5)))))));
    return (0);
}
```

Listing 1: Values of expression.

In the first `printf`(), z gets the value 156 / 14 i.e. 11, y gets the value 22, x is y % 5, i.e. 2, which is the value of the expression output.

In the second `printf`(), x is first assigned the value 5, then z becomes 10, then y becomes 5 % 10, i.e. 5, finally x becomes 15; this is also the value of the expression, which is printed.

In the third `printf`(), z is 5, y is 2, and x is 6, to begin with; then z becomes 13 % x, i.e. 1, x is 3 * z, i.e. 3, and y is 7; then x gets the value z % y, which is 1, finally x becomes 2 * 1, and this also the value output.

3.3.1 Abbreviated Assignment Expressions

It is frequently necessary in computer programs to make assignment such as:

```
n = n + 5;
```

C allows a shorter form for such statements:

```
n += 5;
```

Assignment expression for the other arithmetic operations may be similarly abbreviated:

```
n -=5;   /* is equivalent to n = n - 5; */
n *=5;   /* is equivalent to n = n * 5; */
n /=5;   /* is equivalent to n = n / 5; */
n %=5;  /* is equivalent to n = n % 5;*/
```

The priority and direction of association of each of the operators +=, -+, *=, /= and %= is the same as that of the assignment operator.

E4) Guess the output of the following program.

```
/* Program 3.4; File name:unit3-prog4.c */
#include <stdio.h>
int main()
{
    printf("%d\n", -1 + 2 - 12 * -13 / 14);
    printf("%d\n", -1 % -2 + 12 % -13 % -4);
    printf("%d\n", -4 / 2 - 12 / 4 - 13 % -4);
    printf("%d\n", (-1 + 2 - 12) * (-13 / -4));
    printf("%d\n", (-1 % -2 + 12) % (-13 % -4));
    printf("%d\n", (-4 / 2 - 12) / (4 - 13 % -4));
    return (0);
}
```

Check your answer by compiling the program.

E5) State the output of the following programs:
(Hint: A single `printf()` may be used to print the values of several variables or expression. See Section 3.6 for details. In the `printf()` functions below the first `%d` corresponds to `x`, the second to `y`, the third to `z`, and so on. There must be a one-one correspondence between the format conversion characters (the `%d`s) and the variables in the argument list. Note the commas after the control string, and between identifiers.)

```
/* Program 3.5; File name:unit3-prog5.c */
#include <stdio.h>
int main()
{
    int x = 3, y = 5, z = 7, w = 9;
    w += x;
    printf("w = %d\n", w);
    w -= y;
    printf("w = %d\n", w);
    x *= z;
    printf("x = %d\n", x);
    w += x + y - (z -= w);
    printf("w = %d, z = %d\n", w, z);
    w += x -= y %= z;
    printf("w = %d, x = %d, y = %d\n", w, x, y);
    w *= x / (y += (z += y));
    printf("w = %d, y = %d, z = %d\n", w, y, z);
    w /= 2 + (w %= (x += y - (z -= -w)));
    printf("w = %d, x = %d, z = %d\n", w, x, z);
    return (0);
}
```

```
/* Program 3.6; File name:unit3-prog6.c */
#include <stdio.h>
int main()
{
    int x = 7, y = -7, z = 11, w = -11, s = 9, t = 10;
    x += (y -= (z *= (w /= (s %= t))));
    printf("x=%d, y=%d, z=%d, w=%d, s=%d, t=%d\n",
x, y, z, w, s, t);
    t += s -= w *= z *= y %= x;
    printf("x=%d, y=%d, z=%d, w=%d, s=%d, t=%d\n",
x, y, z, w, s, t);
    return (0);
}
```

E6) Given that `x,y, z` and `w` are `int`s with the respective values 100, 20, 300 and 40, find the outputs from the `printf`()s below:

```
printf("%d\n%d\n%d\n%d", x, y, z, w);
printf("%d\n%d\n%d\n%d", x, y, z, w);
printf("%d%d%d%d%d%d%d%d", x, y, w, z, y, w, z,
x);
printf("%d %d", x + z - y * y, (y - z % w) *
x);
```

3.3.2 Incrementation and Decrementation Operators

Probably the commonest form of assignment to be found in computer programs is of the form:

```
n = n + 1;
```

or

```
n = n - 1;
```

in which a variable `n` is incremented or decremented. Typically `n` may be the index of a loop which is changed by unity in each execution of the loop. From the point of view of program efficiency it is important that such assignments be executed as rapidly as possible, (in fact many modern assembly languages provide for this by including opcodes such as INC or DEC in their instruction sets) and C has special operators, called the **incrementation** and **decrementation** operators for these operations.

The incrementation and decrementation operators (there are two, a **pre-** and a **post-** operator for each operation) are unary operators. They have a priority as high as that of the unary negation operator, and, after the fashion of unary operators, these operators too group from right to left.

The **post-incrementation** operator ++ is written to the right of its operand.

```
n ++;
```

The effect of the statement above is to add one to the current value of `n`; its action is equivalent to:

```
n = n + 1;
```

Besides being more concise, the post-incrementation operation on some computers may be executed faster than this latter form of assignment.

The **post-decrementation** operation is represented by `--` and is written:

```
n --;
```

It decrements `n` by unity, producing a result identical to that of:

```
n = n - 1;
```

The **pre-incrementation** and **pre-decrementation** operators are placed to the left of the operand on which they are to act, and the result of their action is, as before, to increment or decrement the operand:

```
-- n;  /* assigns n - 1 to n */
++ n;  /* assigns n + 1 to n */
```

As far as their operand n is concerned, the pre- and post- incrementation or decrementation operators are equivalent. Each adds or subtracts one to its operand.

Where the pre- and post-operators differ is in the value used for the operand `n` when it is embedded inside expressions.

If it's a "pre" operator the value of the operand is incremented (or decremented) **before** it is fetched for the computation. The altered value is used for the computation of the expression in which it occurs.

To clarify, suppose that an `int` variable `a` has the value 5. Consider the assignment:

```
b = ++ a;
```

Pre-incrementation implies:

```
step 1:   increment a;            /* a becomes 6 */
step 2:   assign this value to b; /* b becomes 6 */
result:   a is 6, b is 6
```

If it's "post" operator the value of the operand is altered **after** it is fetched for the computation. The unaltered value is used in the computation of the expression in which it occurs. Suppose again that `a` has the value 5 and consider the assignment:

```
b = a ++;
```

Post-incrementation implies

```
step 1:   assign the uni-increment a to b ;   /* b becomes 5 */
step 2:   increment a;                        /* a becomes 6 */
result:   a is 6, b is 5
```

The placement of the operator, before or after the operand, directly affects the value of the operand that is used in the computation. When the operator is positioned **before** the operand, the value of the operand is altered **before** it is used. When the operator is placed **after** the operand, the value of the operand is changed **after** it is used. Note in the examples above that the variable `a` has been incremented in each case.

Suppose that the `int` variable `n` has the value 5. Now consider a statement such as:

```
x = n ++ / 2;
```

The post-incrementation operator, possessing a higher priority than all other operators in the statement, is evaluated first. But the value of `n` that is used in the computation of **x is still 5**! Post-incrementation implies: use the current value of `n` in the computation; increment it immediately **afterwards**.

So `x` gets the value 5 /2 = 2, even though `n` becomes 6. We repeat the rule: In an expression in which a post-incremented or post-decremented operand occurs, the current (unaltered) value of the operand is used; then, and only then, is it changed. Accordingly, in the present instance, 5 is the value of `n` that's used in the computation. `n` itself becomes 6.

Now consider:

```
x = ++ n / 2;
```

where n is initially 5.

Pre-incrementation or pre-decrementation first alters the operand n; it is this new value which is used in the evaluation of x. In the example, n becomes 6, as before; but this new value is the value used in the computation, not 5. So x gets the value 6 / 2 = 3.

Let's work through some the assignment statements in the program in Listing 2, to see how x, y, z and w get their values:

```
/* Program 3.7; File name:unit3-prog7.c */
#include<stdio.h>
int main()
{
    int x = 1, y = 2, z = 3, w;
    w = x++ + ++y - --z;
    w = --x - y-- + z++;
    w = x++ * ++y % ++z;
    w = --x / --y * --z;
    w = --x - z -- % --y;
    w = - --x - --y - --z;
    w = ++x * y-- - ++y * x--;
    w = x++ * ++y - x * y * z++;
    return (0);
}
```

Listing 2: Incrementation and decrementation operators.

initially:

x = 1, y = 2, z = 3 and w is undefined.

Consider the first assignment to w:

```
w = x ++ + ++ y - --z;
```

x is post-incremented, so its current value, 1, is used.

y is pre-incremented, so its new value, 3, is used.

z is pre-decremented, so its new value, 2, is used.

Therefore,

$w = 1 + 3 - 2 = 2;$

$x = 2;$

$y = 3;$

$z = 2;$

Let's now look at the second assignment to w:

```
 w = --x - y-- + z++;
```

x is pre-decremented, so its new value, 1, is used.

y is post-decremented, so its current value, 3, is used.

z is post-incremented, so its current value, 2, is used.

Therefore

$w = 1 - 3 + 2 = 0;$

$x = 1;$

$y = 2;$

$z = 3;$

To check if you have understood our discussion of increment and decrement operators, try the following exercises.

E7) Determine the values given to `x, y, z,` and `w` as a consequences of each of the remaining assignment statements in Program 3.7. Verify your answers by inserting appropriate `printf`()s in the program, and executing it.

E8) Give the output of the following program:

```
/* Program 3.8; File name:unit3-prog8.c*/
#include <stdio.h>
int main()
{
    int x = 10, y = 11, z = 12, w;
    w = ++x - y++;
    printf("w = %d, x = %d, y = %d\n", w, x, y);
    w = ++z % - --y;
    printf("w = %d, z = %d, y = %d\n", w, z, y);
    w = ++y + x++ * z--;
    printf("w = %d, y = %d, x = %d, Z = %d\n",
w, y, x, z);/*Continued*/
    w = ++x % ++y % ++z % w--;
    printf("w = %d, x = %d, y = %d, Z = %d\n",
w, y, x, z);/*Continued*/
    w = ++w / ++x / y--;
    printf("w = %d, x = %d, y = %d\n", w, x, y);
    return (0);
}
```

In the next section, we will explain the concept of lvalue and rvalue of a variable. This will give you better insight into the way that expressions are evaluated in computer.

3.4 LVALUES AND RVALUES

One question that might have occurred to you after reading the foregoing is: what meaning, if any, is to be associated with a statement such as `-n++;`? The answer is, none whatever; in fact such statements will elicit from the compiler vehement protest, and it's instructive to see why. But before that we'll need to take a quick look at the way a computer stores instructions and data, and executes programs.

As you know, the **core** or **main** memory of a computer, also called its **RAM**, (the abbreviation for Random Access Memory) is divided into units known as **words**. We've also seen in the foregoing that an **int** variable, and in many cases even a **short**, usually occupies a word of memory, a **long int** may occupy two words, a **double** may occupy four, and so on.

Depending on the computer a word of memory may be two, four or even eight bytes big. (For the sake of accuracy, however, it must be stated that there have been computers, in fact very popular computers, whose word lengths were not integral multiples of 8 bits: for example, the 36-bit DECSYSTEM-10 and DECSYSTEM-20 machines from Digital Equipment corporation, U.S.A.) In machines with large word sizes the gradation between **short**s, **int**s or **long**s or between **float**s and **double**s may be different than indicated above. As we've seen, each word has associated with it a unique address, which is a positive integer that helps the CPU to access the word. Addresses increase consecutively from the top of memory to its bottom. When a program is compiled and linked, each instruction and each item of data is assigned an address. At execution time instructions and data are found by the **CPU** from these addresses.

The **PC**, or **Program Counter**, is a CPU **register** which holds turn by turn the addresses of successive instructions in a program. In the beginning the PC holds the address of the zeroth instruction of the program. The CPU fetches and then executes the instruction to be found at this address. The PC is meanwhile incremented to the address of the next instruction in the program. In computer jargon, that is where the PC now **points**. Having executed one instruction, the CPU goes back to look up the PC. Therein, in its updated value, it finds the address of the next instruction in the program. This instruction may not necessarily be in the next memory location. It could be at a quite different address, a random distance away from the one just executed (For example, the last statement could have been a **goto** statement, which unconditionally transfers control to a different point in the program; or there may have been a branch to a function subprogram.). The CPU fetches the contents of words addressed by the PC in the same amount of time, whatever their physical locations. That, we recall, is why core memory is called “random access memory”. The CPU has random access capability to any and all of the words of memory, no matter what their addresses may be.

note

In time sharing computers with virtual memory this may not be a quite accurate picture. But that’s another story.

Program execution proceeds in this way until the last instruction has been processed by the CPU.

Now, the names of program variables, too, are associated with memory addresses; one of the functions of the compiler is to prepare a **symbol table** which contains the memory address of each program variable. Whenever there’s a reference to a variable in the program, the CPU is directed to the address from where it can fetch its value. While HLL programmers refer to variables in their programs, the CPU knows only the addresses of memory at which it can find them; in other words when we refer to a variable `n` in a program, the CPU looks at the memory address where it is stored.

The address associated with a program variable in C called its **lvalue**; the contents of that location is its **rvalue**, the quantity which we think of as the value of the variable. The notation corresponds with our picture of memory, a table with two columns in which the left hand column contains memory addresses, and the right hand column the contents of those addresses. The rvalue of a variable may change as program execution proceeds; its lvalue, never. The distinction between lvalues and rvalues becomes sharper if you consider the assignment operation with variables `alpha` and `beta`:

```
alpha = beta;
```

`beta`, on the right hand side of the assignment operator, is the quantity to be found at the address associated with `beta`, i.e. is an rvalue, the contents of the variable `beta`, `alpha`, on the left hand side, is the address at which the contents are altered as a result of the assignment. `alpha` is an lvalue. The assignment operation deposits `beta`’s rvalue at `alpha`’s lvalue. Think of an lvalue as something to which an assignment can be made.

The incrementation and decrementation operators operate on lvalues, and increment or decrement their rvalues. It should now be clear why `-- n ++`, which may be interpreted as `--(n = n + 1)`, is not an allowed operation: `n ++` is an expression, not an lvalue, not an object to which an assignment can be made. For the same reason `(-n) ++` is not correct, but

```
m = -n ++
```

is syntactically valid, since the right to left associativity of the unary operators involved ensures that the expression is evaluated in the order

```
m = - (n ++),
```

i.e.

```
m = -(n), /* use the unincremented n */
n = n + 1./* then increment it */
```

What happens when there are two variable of different sizes in expression? What if the size of a variable is too small to hold the result of an expression? These questions are answered in the next section.

3.5 PROMOTION AND DEMOTION OF VARIABLE TYPES: THE CAST OPERATOR

We've seen that the fundamental types of C variables fall broadly into two categories, integer and floating point, with each category further classed according to size: **char**, **short**, **int**, **long** and **float**, **double** and (in ANSI C) **long double**. C is fairly permissive that it allows the assignment of rvalues of one type to lvalues of a different type: Thus, so far as the syntax of the language is concerned, it's okay to assign a **char** value to an **int** variable, and vice versa. But it's important to be aware of the ramifications of such assignment; for when a "wider" type such as **float** is assigned to a less wide type such as **int** or **char**, one is really trying to do the impossible: Squeeze a big object into a small and in this case a fundamentally "differently shaped" hole: because floating point numbers are stored and operated upon quite differently from integer types. This, in C parlance, is called a **demotion**. Some bits are going to get knocked out. Which will they be? How can the resulting assignment be interpreted? Contrarily, when a less wide type is assigned to a type of larger width, such as in the assignment of a **char** to an **int** (this is called a **promotion**, and it's certainly a less traumatic operation than demotion), how are the extra bits filled: By ones or by zero?

The conversion of data types is subject to a few rules, which we'll explore through the programs below.

i) **The Conversion of Floating point Numbers to Integers.** Let's first look at the conversion of a floating point value to one of an integer type, say **int**. The operative rule is this: If the value is small enough for it to fit into an **int**, the digits after its decimal point are discarded, and the **int** variable is assigned the integer part of the floating point number. But if the value of the **float** variable is negative, then the truncation may be towards zero on some machines, and away from zero on others. For example, if -6.78 is assigned to an **int** variable, the value of the latter may be -6 on some machines, -7 on others. It is not defined in the language as to what may happen if a negative floating value is assigned to an **unsigned int**, or if a floating number or magnitude greater than the limit of **int** is assigned to an **int** variable. The program in Listing 3 was executed in VAX C (where **int**s occupy four bytes, while **short**s are two byte quantities): observe that its results are in accordance with these rules. Observe also that the results of a demotion cannot be predicted.

```
/* Program 3.9; File name: unit3-prog9.c */
#include <stdio.h>
int main()
{
    char one_byte;
    short int two_bytes;
    int small_box;
    double large_value = 12.34e+24;
    float neg_value = -2.78964e+8;
    float small_neg_val = -6.78;
    small_box = large_value;    /* double demoted to int */
```

```
    printf("large_value %e in small_box (4 byte int); %d\n",
large_value, small_box);/*Continued.*/
    two_bytes = neg_value;      /* negative float to short */
    printf("neg_value %e in two_bytes: %d\n",
neg_value, two_bytes);/*Continued*/
    one_byte = small_neg_val;   /*small neg. float to char */
    printf("small_neg_val %f in one_byte: %d\n",
small_neg_val, one_byte);/*Continued*/
    return (0);
}
```

Listing 3: Conversion of floats to ints.

```
/* Program 3.9 :  Output :  */
large_value 1.234000e+25 in small_box (4 byte int):
1073741824
neg_value -2.789640e+8 in two_bytes:  22752
small_neg_val -6.780000 in one_byte:  -6.
```

On a PC with an ANSI C compiler execution was aborted with the error message:

```
Floating point error:  Overflow.
Abnormal program termination
```

The output of the program in Listing 3 should warn you that the results of demotion of type will in general by utterly unpredictable, and should you use such demotions in your programs, do so with the full consciousness that you're doing so!

E9) Experiment with the program in Listing 3 on your system, and study its output using variables of differing widths, types and values.

---

ii) **The conversion of doubles to floats, and vice-versa.**

When a **double** value is converted to a **float**, the value is rounded to the precision of **float**, before truncation occurs. If the result is out of range, the behaviour is undefined. In general, floating point arithmetic is carried out in double precision. When a variable of type **float** is converted to **double**, its fractional part is padded with zeros.

ANSI C

If a **float** value is followed by an `f` or `F`, it's treated as a single precision number, and is NOT converted to **double** in computation; a floating point value followed by an `l` or `L` is treated as **long double**.

```
/* Program 3.10; File name: unit3-prog10.c */
#include <stdio.h>
int main()
{
    double double_width = 12345.0123456789;
    float single_width;
    single_width = double_width;
    printf("double_width = %16.10f, single_width = %f\n",
double_width, single_width);    /*Continued */
    return (0);
}
```

Listing 4: Conversion of doubles to floats and vice-versa.

The output of the program in Listing 4 on our system was:

```
double_width = 12345.0123456789
single_width = 12345.012695
```

The `%16.10f` format conversion specifier for `double_width` ensured that its value was output in a minimum of 16 columns, precise to 10 decimal places. The value of `sinlge_width` was rounded to the precision of **float**. Note that the value is correct to only eight significant figures. See Section 3.6

iii) **The Promotion and Demotion of Integer Types**

When a signed integer type is promoted to an integer type of larger width, the sign is extended to the left. For example, the representation of the two-byte **int** −1 on a two's complement machine is:

1111111111111111 (hex FFFF)

If this value is promoted to **long**, the sign is left-extend, so that its representation as a four byte quantity becomes a collection of 32 one bits:

FFFFFFFF

This is illustrated in the output of the program in Listing 5 below.

Sign extension with one bits does not occur when an **unsigned int** or **char** is assigned to a wider type.

When a integral type is coerced to a less wide **int** type variable the leftmost bits of the wider type are truncated, provided both source and destination variables are **unsigned**. But if the source is a signed quantity, it is accommodated without change in the destination variable if its magnitude is within the limit of the destination's type, otherwise the result is undefined.

The unary **cast** operator (type_cast) of C, introduced in Program 3.11 below is very useful when you need to convert from a variable or expression of one type to one of another. It works like this: suppose `alpha` is an **int** variable, then (**long**) `alpha` is an expression of type **long**, (**float**) `alpha` casts the value of `alpha` to the type **float**, (**short**) `alpha` demotes its value to a `shortint`, (**double**) `alpha` promotes it to type **double**, and so on. The operand of a cast operator may be an rvalue or an expression. The cast operator **does not** alter the rvalue or the type of its operand. The expression (type_cast) rvalue acquires the type enclosed in the parentheses only for the current computation. The cast operator has a priority just below the parentheses operator, and groups from right to left.

```
/* Program 3.11; File name: unit3-prog11.c */
#include <stdio.h>
int main()
{
    short alpha = -5, beta = 5;
    long lambda = 12345678L, mu = -12345678L;
    printf("alpha cast as a long = %ld, \
in hex = %lx\n", (long) alpha, (long) alpha);
    printf("\nNote appropriate sign extension...\n");
    printf("\nbeta cast as a long = %ld, \
in hex = %lx\n", (long) beta, (long) beta);
    printf("\nlambda cast as an \
int = %d, as a char = %d\n", (int) lambda, (char) lambda);
    printf("\nmu cast as an \
int = %d, as a char = %d\n", (int) mu, (char) mu);
    return (0);
}
```

Listing 5: Promotion and demotion of integer types.

Here's the output of program in Listing 5:

```
/* Program 3.11 :  Output:  */
alpha cast as a long = -5, in hex = fffffffb
Note appropriate sign extension...
beta cast as a long = 5, in hex = 5
lambda cast as an int = 24910, as a char = 78
mu cast as an int = -24910, as char = -78
```

Verify that (**int**) `lambda` and (**char**) `lambda` are indeed the contents of the leftmost two bytes, and the leftmost byte, respectively, of the **long int** variable `lambda`.

In the evaluation of an expression involving more than one operand data type conversions obey the following rules in ANSI C:

a) All **char** and **short int** values are elevated to **int**. After this, the following conversions are done operation by operation as follows:

b) If one of the operands in an operation is **long double**, the second operand is also converted to **long double**; otherwise

c) if one of the operands is a **double**, the other operand is converted to **double**; otherwise

d) if one of the operands is a **float**, the second is also converted to **float**; otherwise

e) if any one operand is **unsigned long**, the other operand is also converted to **unsigned long**; otherwise

f) if any one operand is **long**, the other operand is converted to **long**; otherwise

g) if any one operand is of type **unsigned**, the other operand is converted to **unsigned**. Only remaining possibility is that

h) both operands must be **int**s, and that is also the type of the result.

i) If one operand is **long** and the other is an **unsigned int**, and if the value of the **unsigned int** cannot be represented by a **long**, both operands are converted to **unsigned long**.(For example, if you have solved exercise 6 of unit 2, you would have most probably found out that for the gcc compiler on a 32-bit machine, the maximum size of long is 2147483647 and the the maximum size of **unsigned int** is 4294967295, the same as **unsigned long**. Confusing? Remember that the limits in Table 1 of Unit 2 prescribe the **minimum magnitudes** and compilers can allow larger values! So, on a 32-bit machine, the size of **unsigned int** can be larger than **long**! It is for handling this situation that rule viii) is prescribed.)

Let's apply these rules to the evaluation of an expression involving a **float** variable `floatex`, and **int** variable `intex`, a long variable `longex` and a **short** variable `shortex`:

```
floatex * shortex + longex % intex
```

By rule i) `shortex` is promoted to **int**. Then, in the evaluation of

```
floatex * shortex
```

`shortex` is promoted to float by rule iv). In the evaluation of :

```
longex % intex
```

`intex` is promoted to **long** by rule vi), and the expression itself though of magnitude less than `intex`, is of type **long**.

In the computation of the entire expression, which consists of an expression that is **float** and one that is **long**, rule iii) makes the **long** expression a **float**, and the result is also **float**.

Example 2: Write a C programme that solves the following problem: Pradeep's car travels 238 kilometres on 15 litres of petrol. How many kilometres does that average to per litre?

It wouldn't do simply to declare the variables `distance_travelled` and `litres_consumed` simply as **int**s, and perform integer division; for the quotient would be truncated to its integer part, and the value would be off by quite a **bit**, causing avoidable disquietude in Pradeep's mind. But we have now at lest two ways to solve this problem: either to declare one `distance_travelled` or `litres_consumed` a **float** variable, or to declare both as **int**s, and use the cast operator to convert the value of any variable to **float**. The program in Listing 6 on the next page below uses the latter approach.

```
/* Program 3.12; File name:unit3-prog12.c */
#include <stdio.h>
int main()
{
    int distance_travelled = 228, litres_consumed = 15;
    printf("Pradeep's car travels %f kilometers per litre.\n",
(float) distance_travelled / litres_consumed);
    return (0);
}
```

Listing 6: Average petrol consumed per litre.

Here's the output of our little program, a little more satisfying for Pradeep than the value that would have been obtained without using the cast operator:

Pradeep's car travels 15.866667 kilometre per litre.

* * *

Try the following exercise now.

E10) Modify the above program so that it's a little more useful for vehicle owners: the program should prompt for the initial and final readings on the milometer, and the petrol consumed for the distance travelled (which is the difference of these readings), and should then output the average obtained per litre.

We close the section here. In the next section, we will discuss format control in `printf()` and `scanf()` functions in detail.

3.6 FORMAT CONTROL IN THE `printf()` AND `scanf()` FUNCTIONS

We have seen that the `printf()` function can be used to output strings of characters as well as numbers; the fact is that `printf()` is a versatile function which can be used to deliver the formatted output of any numbers of variables or expression of any type. Some of the capabilities of `printf()` are illustrated in the examples below.

When `printf()` is used to output the values of variables or of expressions formed from them, the formatting information is supplied in its first argument, which is a string called the **control string**. The variables to be printed are listed after the control string. Commas are required to separate the control string and the variables listed. A `print()` that's used to output the values of variables looks typically like this

`printf("control string", var_1, var_2, ..., var_n)`

Format specification, i.e. how the variables to be output should be displayed on the screen, is provided in the control string; var_1, var_2,..., var_n are variables or expressions. If there are no variables to be printed, as in our early examples of `printf()`, the control string must not contain any format specifiers. In this case the string itself is output:

```
printf("This string does not refer to a variable list.");
```

Let's now suppose that you wish to print the values of two **int** variables, `x` and `y`, which are respectively 27 and 117. Then the following statement will do:

```
printf("%d %d \n", x, y);
```

The first `%d`—called a **format conversion specifier**—in the control string instructs that the variable `x` will be printed as a **decimal integer** (The `d` stands for decimal.) in a **field**

(i.e. consecutive spaces on a line) as wide as may be necessary to accommodate it. Becasue x is a two-digits number, it will be printed in field of width two units.

The second %d refers to the variable y, which, being a number of three digits, will be printed in a field three units wide.

Note that there are two variables to be printed, x and y, and there are two %d's in the control string, one for each variable to be output.

The output in the present case will be :

```
27 117
```

Observe that there is but a single space separating the values of x and y in the output. This is because we had placed a single space between the %d's in the control string. Suppose instead that we choos the following format specification, which included the escape sequence for a tab between the %d's:

```
printf("\%d\t\%d\m", x, y);
```

The values of x and y will then be separated by a tab in the output:

```
27 117
```

Any characters other than the format conversion characters such as %d, etc. within the control string are reproduced directly in the output. So spaces, tabs, escape sequences and text in the control string appear without change on the monitor. Therefore the output of:

```
printf("\t The value of x = \%d,\n\t while y = \%d.", x, y);
```

will be

```
The value of x = 27,
while y = 117.
```

printf() may similarly be used to output expressions; suppose that x and y have the values 27 and 117 as before. Then the statement:

```
printf("\%d \%d\n", y / x, y, \% x);
```

gives the following output:

```
4 9
```

It is very important that there should be a one-one ordered correspondence between each format conversion specifier in the control string, and the list of variables or expressions which follows it. If you wish to print twenty **int** variables: var_1, var_2,..., var_20, there should be twenty occurrences of the %d format conversion specifier within the control string, the first assoicated with the first value to be output, the second with the second, and so on.

Because the % character has a syntactic value in the control string, two consecutive occurrence % % are required in the string in order to print it as a literal in the output. Let's look at the example in Listing 7:

```
/* Program 3.13; File name: unit3-prog13.c */
#include <stdio.h>
int main()
{
    int chance_of_rain = 70;
    printf("There\'sa %d%% chance of rain today.\n",
chance_of_rain);/*Continued*/
```

```
    return (0);
}
```

Listing 7: Printing % character in `printf()`

The two % %'s after the %d are output as a single % character. Verify this by executing the program.

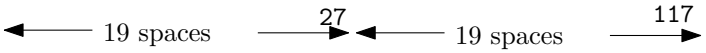
It is often desirable to print a value right or left justified within a specified field width, say w. (For example, when a Rupee amount is to be printed on a cheque, common sense dictates it should be printed left justified in the designated field.) %wd right justifies the value to be output within a field of width w, %-wd left justifies within the field.

To print the values of x and y (which we assume as before are 27 and 117 respectively) right justified in fields of width 19, use:

```
printf("The values of x and y are:\n\n\%19d\%19d\n", x, y);
```

Output:

```
The values of x and y are:
```

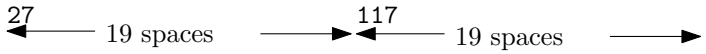


To print the values of x and y left justified in fields of width 19, we write:

```
printf("The values of x and y are:\n\n\%-19d\%-19d\n", x, y);
```

Output:

```
The values of x and y are:
```



If the first digit of the field width is zero, the field is stuffed with leading or trailing zeroes, depending on whether the justification is towards the right or towards the left.

If a field width is not specified, or if the width specified is insufficient to accommodate all the digits in the number, %d still outputs the correct value. Therefore we will frequently not specify a field width with a format specification.

The %0 and %x format conversion characters are used to output octal and hexadecimal integers repectively.

The %u specification is used to print unsigend **int**s, and %d, %l0, and %lx specifiers are used to output **long**s as decimal, octal or hex integers respectively.

E11) Do you agree with the output of the program below?

```
/* Program 3.14; File name: unit3-prog14.c */
#include <stdio.h>
int main()
{
    typedef int debt;
    debt amount = 7;
    printf("If I give you Rs. %05d\n", amount);
    printf("you will owe me Rs. %-05\n", amount);
    return (0);
}
```

E12) State the output of the following programs:

```
/* Program 3.15; File name: unit3-prog15.c */
#include <stdio.h>
int main()
{
    int birth_hour = 23, birth_minute = 47;
    int birth_day = 17, birth_month = 3, birth_year = 1981;
    printf("\t Abhishek was born on %02d-%02d-%4d\n",
birth_day,birth_month, birth_year);/*Continued*/
    printf("at %02d:%02d hours\n", birth_hour, birth_minute);
    printf("in the city of Jodhpur, Rajasthan.\n");
    return (0);
}

/* Program 3.16; File name: unit3-prog16.c */
#include <stdio.h>
int main()
{
    int amount = 64;
    /* the 64,000 Dollar question */
        printf("Q: How can you make 64000 dollars from 64?\n");
    printf("A: By clever printing: %-05d", amount);
    return (0);
}
/* Program 3.17; File name unit3-prog17.c */
#include <stdio.h>
int main()
{
    int x = 27, y = 117;
    printf("y %% x = %0 in octal \
and %x in hex", y % x, y % x);
    return (0);
}
/* Program 3.18; File name: unit3-prog18.c */
#include <stdio.h>
int main()
{
    unsigned cheque = 54321;
    long time = 1234567L;       /* seconds */
    printf("I\'ve waited a long time (%1d seconds)\n", time);
    printf("for my cheque (for Rs. %u/-), and now\n", cheque);
    printf("I find it\'s unsigned!\n");
    return (0);
}
```

3.6.1 Floating point Numbers and Character Strings with printf()

The output of **float**s and **double**s is accomplished by the `%f` specifier where an optional `w.d` is used to specify a field width `w` and number of digits, `d`, after the decimal point. `%e` is used to output single or double precision floating point numbers in scientific notation (e.g. 1.234e5) and `%g` prints a float value either as `%e` or as `%f`, whichever is shorter. The width and precision specification are optional, as before. A precision specification rounds off the value that is output to the number of places stated.

C strings are output directly via the `printf()`; however the `%s` conversion character with an optional `w.d` specification can be used to format the output. In this case `d` number of characters from the beginning of the string will be printed in a field of width `w`.

Example 3: Let us look at an example program. The program in Listing 8 on the next page prints the values of the Sine function for 0°. 10°, . . ., 90°. Another thing you

would notice is that we have converted degrees to radians; This is because the Sin() function takes input in terms of radians. Here we have used the **for** loop. You may be able to figure out how it works by compiling the program, running it and looking at the output. You may also look at Unit 6 for a description of the **for** loop.

```
/*Program to print the values of the Sine Function.*/
/*File name:unit-3-sinetable.c*/
#include<stdio.h>
#include<math.h>
#define PI 3.1416
int main()
{
    int i;
    printf("%-7.5s\t%-14.12s\n\n", "THETA", "SIN(THETA)");
    for (i = 0; i <= 90; i += 10)
        printf("%-7d\t%-14.4f\n", i, sin((PI * i) / 180.0));
    return (0);
}
```

Listing 8: A program to print the values of the Sine function.

You may experiment with different format control statements and look at the output.

* * *

Try the following exercises to check your understanding of format control of floating point and string variables.

E13) Execute the following program to verify the rules stated above for the output of floating point variables"'

```
/* Program 3.19; File name: unit3-prog19.c */
#include <stdio.h>
int main()
{
    double pi = 3.14159265;
    printf("%15f\n", pi);
    printf("%15.12f\n", pi);
    printf("%-15.12f\n", pi);
    printf("%15.4f\n", pi);
    printf("%15.0f\n", pi);
    printf("%15.3g\n", pi);
    printf("%15g\n", pi);
    printf("%15.4e\n", pi);
    printf("%15e\n", pi);
    return (0);
}
```

E14) What does the following program print?

```
/* Program 3.20; File name: unit3-prog20.c */
#include <stdio.h>
int main()
{
    printf("%-40.24s", "Left justified printing.\n");
    printf("%-40.20s", "Left justified printing.\n");
    printf("%-40.16s", "Left justified printing.\n");
    printf("%-40.12s", "Left justified printing.\n");
    printf("%-40.8s", "Left justified printing.\n");
    printf("%-40.4s", "Left justified printing.\n");
    printf("%-40.0s", "Left justified printing.\n");
    printf("%40.25s", "Right justified printing.\n");
    printf("%40.20s", "Right justified printing.\n");
    printf("%40.15s", "Right justified printing.\n");
```

```
    printf("%40.10s", "Right justified printing.\n");
    printf("%40.5s", "Right justified printing.\n");
    printf("%40.0s", "Right justified printing.\n");
    printf("%40.0s", "Right justified printing.\n");
    return (0);
}
```

We close this section here. In the next section, we will summarise our discussion in this unit.

3.7 SUMMARY

In this unit, we have studied the following:

1) The precedence and associativity of operators: `()`, `[]`, `->` and `.` have the highest precedence and associate left to right. Unary operators like `!`, `~`, `++` , `--`, `+` have the next highest precedence and associate right to left. The operators `*` and `/` have the next highest precedence and associate left to right.
2) C expressions are syntactically valid combinations of operators and operands that compute to a value determined by the priority and associativity of the operators.
3) Incrementation and decrementation operators and abbreviated assignment operators `+=`, `-=`, `*=` and `/=`.
4) The format modifier `%wd` prints a integer right justified in a field of width `w` and the modifier `%-wd` prints an integer left justified in a field of width `w`.
5) The format modifier `%w.d` prints a floating point number in a field of width `w` with `d` decimat digits after the decimal point.
6) The format modifier `%w.ds` prints `d` characters from the beginning of a string in a field of width `w`.

3.8 SOLUTIONS/ANSWERS

E1) We have dealt with one case, a is negative and b is positive. Please add code for the remaining 3 cases.

```
/*Prog 3.0. Filename:unit3-prog0.c*/
#include<stdio.h>
int main()
{
    int a, b, ans1, ans2;
    /* a is negative and b is positive. */
    a = -10;
    b = 5;
    ans1 = a % b;
    ans2 = a - (a / b) * b;
    printf("The difference is %d\n", ans1 - ans2);
    return 0;
}
```

E2) In the first set the values are x = 7, y = 62 and z = 1. The others are easy. In the case of z, 2*3*4 is 24 and 24%15 is 1. 1%13 is again 1. You can similarly check the other values. Here is the program with `printf()` statements added. You can check your answers by compiling and running the program.

```
/* Program3.1; File name:unit3-ansex2.c */
#include<stdio.h>
int main()
{
    int x, y, z;
    x = 2 + 3 - 4 + 5 - (6 - 7);
    y = 2 * 33 + 4 * (5 - 6);
    z = 2 * 3 * 4 / 15 % 13;
    printf("x is %d, y is %d, z is %d\n",x,y,z);
    x = 2 * 3 * 4 / (15 % 13);
    y = 2 * 3 * (4 / 15 % 13);
    z = 2 + 33 % 5 / 4;
    printf("x is %d, y is %d,  z is %d\n",x,y,z);
    x = 2 + 33 % -5 / 4;
    y = 2 - 33 % -5 / -4;
    z = -2 * -3 / -4 % -5;
    printf("x is %d, y is %d,  z is %d\n",x,y,z);
    x = 50 % (5 * (16 % 12 * (17 / 3)));
    y = -2 * -3 % -4 / -5 - 6 + -7;
    z = 8 / 4 / 2 * 2 * 4 * 8 % 13 % 7 % 3;
    printf("x is %d, y is %d,  z is %d",x,y,z);
return (0);
}
```

E3) Since x = 3, y = 5 and z = 7 the first expression

`w = x % y + y % x - z % x - x % z;`

is `3%5+5%3-7%3-3%7`. Since % has higher priority, the value is $3+2-1-3=1$. You can similarly find the value w in the other cases. Do not forget to check your answer by compiling and running the program.

E4) Since * and / have higher priority and they group left to right, first `-12*-13` is computed, which is 156. Since `156/14=11`, we get `-1+2+11=12`. You can similarly find the value of the other expressions.

E5) In program 3.5, after executing the statement `w +=x`, the value of `w` is $9+3=12$ and this is printed by the first `printf()`. After executing the statement `w -= y`, the value is $12-5=7$. You can similarly work out the other values. In program 3.6, `s %= t`, the value of `s` remains 9, because `9&10=9`. This is the value of the expression `s %= t`. Since `w` is -11, the value of the expression `w /= (s %= t)`, i.e. `-11/9` is -1. So, the value of the expression `(w /= (s %= t))` is -1. Since `z` is 11, the value of the expression `z *= (w /= (s %= t))` is $11\times-1=-11$. So, `y -= (z *= (w /= (s %= t)))` is $y-(-11)=y+11=4$. So, the value of the expression `x += (y -= (z *= (w /= (s %= t))))` is $7+4=11$. Similarly, you can find the value of the other expression.

E6) In the last `printf()`, $x+z-y*y$ is $100+300-400=0$ and `(y - z % w) * x)` is `(20-300%40)*100)` is 0 since 300%40 is 20. So, the values printed will be 0, 0.

E7) After evaluating the first 2 statements, the values are `x` = 1, `y` = 2, `z` = 3. While evaluating the expression `w = x++ * ++y % ++z`, `x` is post-incremented and the value `x` = 1 is used. The variable `y` is pre-incremented and the value `y` = 3 is used. The variable `z` is pre-incremented and `z` = 4 is used. So, `w = 1 * 3 % 4` is 3. After evaluating the expression, the value of `x`, `y` and `z` are `x` = 2, `y` = 3, `z` = 4.

In the expression `w = --x / --y * --z`, `x` is pre-decremented and the value `x` = 1 is used. The variable `y` is pre-decremented and the value `y` = 2 is

used. The variable z is pre-decremented and the value z = 3 is used. So, w = 1 / 2 * 3 = 0. After evaluating the expression, the value of x, y and z are x = 1, y = 2, z = 3. Similarly you can check the output of the remaining programs.

E8) The valus x = 11 and y = 11 are used in evaluating the expression w = ++x - y++. The value of w is 0 and the value of x and y are 11 and 12 after the statement is exectued. Similarly, you can find the other values. Check your answers by compiling and running the problem.

E10)
```
#include<stdio.h>
int main()
{
    int dist_travelled, pet_consumed, ini_reading, fin_reading;
    printf("Enter the initial reading...:\n");
    scanf("%d", &ini_reading);
    printf("\nEnter the final reading...:\n");
    scanf("%d", &fin_reading);
    printf("\nEnter the petrol consumed..:\n");
    scanf("%d", &pet_consumed);
    dist_travelled = fin_reading - ini_reading;
    printf("Pradeep's car travels %f kilometers per litre.\n",
(float) dist_travelled / pet_consumed); /*Continued */
    return (0);
}
```

UNIT 4 DECISION STRUCTURES IN C

Structure **Page No.**

4.1 INTRODUCTION

Recall that, in Unit 1, we saw a program for solving quadratic equations. There, we have to find out the whether the roots are equal, real or imaginary. This in turn depended on the fact whether the discriminant was zero, positive or negative. For each case, we had to take a different course of action. We did this using the **if**() statement. In this Unit, we will discuss the **if**() and its extension **if**()-**else** statements of C. In the program, we also checked whether the discriminant is positive using the statement **if** (disc > 0). The expression disc > 0 is an example of a **Boolean expression**. We begin our unit with a discussion on Boolean expressions in Sec. 4.2. In Sec. 4.3, we discuss the **goto** statement. In Sec. 4.4, we discuss the **if**() statement. In Sec. 4.5, we discuss **if**()-**else** statement.

Objectives

After studying this unit, you should be able to

- use the six Boolean operators of C;
- use the operator for negation;
- use the logical connectives for **and** and **or**;
- create programs with branches; and
- use the (disreputable) **goto** statement.

4.2 BOOLEAN OPERATORS AND EXPRESSIONS

As we mentioned in the introduction, we used the **if**() in the program to solve the quadratic equation, that we had given in Unit 1. In the **if**() statement the parentheses which follow the keyword **if** contain a special type of C expression, called a **Boolean expression**, which evaluates to one of two values—**true** or **false**, but nothing in between. Booleans are black or white, there are no shades of grey! For example, the Boolean expression **if**(disc > 0) evaluates to either true or false depending on whether the discriminant is positive or not. Typically Booleans compare the values of two variables or expressions, through **relational operators**; these are used to construct expressions that evaluate to **true** or **false**. The result of a Boolean expression can thus be used to guide the flow of control in a program to alternative courses of action: do such if the Boolean is **true**, such other if **false**.

Consider, for example, a program that a small bank might use for its book-keeping, to permit a customer to encash a cheque if the amount she wishes to withdraw is less than her balance:

```
if (amount_requested < current_balance)
   printf("Issue cash, Rs%-d\n", amount_requested);
else
   printf("Sorry, amount requested exceeds balance.\n");
```

The expression

```
amount_requested < current_balance
```

is an example of a Boolean expression. Obviously, it can only be **true** or **false**. Either the value of `amount_requested` is less than `current_balance`, or it's not. If it's less, issue cash, else not.

Suppose `x` and `y` are `int` variables that have the respective values 3 and 5. Then the Boolean expression:

```
x > y  /* Is  x greater than  y? */
```

has the value **false**, while

```
x < y & /* Is  x <  y? */
```

has the value **true**, and

```
x == y  /* Is x equal to  y? */
```

is **false**$<, >$ and == are **relational operators**. Boolean expressions such as

```
amount_requested < current_balance
```

are generally (but in C not necessarily, as we shall find), constructed from the relational operators. C has three other relational operators:

```
>=   /* greater than or equal to */
>=   /* less than or equal to */
!=   /* not equal to */
```

In addition there is the unary **negation operator**, !, which negates the expression on its right. The priorities and associativities of the relational operators are listed in Table 3.1.

In an `if ()` statement the parentheses containing the Boolean are followed by a statement called the **object** statement of the `if ()`; the object statement is executed if the parenthetical condition is **true**, and is ignored otherwise. In the last example, the statement:

```
printf("Issue cash, Rs.%-d\n", amount_requested);
```

is the object statement of the associated `if ()`.

The `if ()` statement imparts decision-making power to a program. The object statement is executed if and only if the value of the Boolean expression so warrants:

```
if  (this condition is true)
execute this object statement, otherwise ignore it;
```

Example:

```
if (x > y)
printf("x is greater than y.\n");
```

A variant of the `if ()` statement is the **if ()-else** statement, which has an object statement with the `if ()` part, and another with the **else** part: if the parenthetical condition evaluates to **true**, there's money in the account, allow a withdrawal; else not. Example:

```
if  (x + 3 < Z * W)
printf("x plus 3 is less than z times w.\n");
   else
printf("x plus 3 is not less than z times w.\n");
```

Note from Table 3.1 that the priority of < is less than that of the arithmetic operators. So the expression:

```
x + 3 < z *w
```

is evaluated in order `z * w, x + 3.  x + 3 < z * w`.

We use **if** () and **if** ()-**else** statements in our daily lives. Here are two examples from mine:

Example:

```
if  (it's a pleasant day)
   printf("I think I\'ll relax today.");
```

Example:

```
if  (it's a pleasant day)
   printf("I think I\'ll relax today.\n");
else    /* because it's not a pleasant day */
   printf("Sorry, I won\'t be able to work today.\n");
```

(You'll be unlikely to catch me working most days of the week, whether the days are pleasant or not!)

Such statements provide the computer, as they do us, with the almost magical power to make decisions; and bring into sharp focus the main difference between a hand-held calculator and a computer: a calculator cannot execute an **if** () statement; it cannot decide between alternative course of action, depending on the result of a computation. Of course, neither can it store a program in its memory.

Computers can demonstrate amazing feats of what appears at first sight to be "reasoning behaviour." (Whether this apparently intelligent behaviour is in fact akin to a form of human begin-like reasoning is the subject of intense debate among Artificial Intelligence experts). In substantial measure the "intelligence" that computers seem to exhibit is due to the fact that the internal two-valued logic of the CPU is isomorphic to the Boolean logic used almost instinctively by intelligent human beings.

For a first glimpse of Boolean logic, consider the following problem:

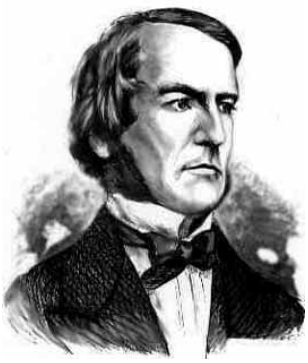
Amal will join the Good Men's Club if either Bimal is chosen its President or Saran its Vice-President.

Bimal will accept Presidentship of the Club only if Ashok is made Vice-President and Rajni **not** made the Club Secretary.

Rajni will not join the Club if Suman opts for membership.

Saran decides not to join the club, Ashok is made Vice-President, and Suman becomes a club members.

What will Amal do?



George Boole
1815–1864

Logic problems such as this one are most easily solved by the techniques of an algebra called **Boolean Algebra**, in honour of its inventor, George Boole. The "variables" of Boolean Algebra are expressions which can have one of precisely two values: they can be either **true** or **false**. (For this discussion we will use the words expression and statement interchangeably.)

Thus, it's either **true** or **false** that "Amal will join the Good Men's Club". There is no third alternative. The problem is to determine the **truth value** of this statement, given the truth values of the statements with which it is connected.

From the information supplied, the statement acquires the value **true** if Bimal is made President of the Club **or** if Saran becomes its Vice-President. In either case, Amal will join the Club. The **or** here is an example of a **logical connective**. It connects two Boolean variables (or constants).

Each of the statement:

Bimal is President
Saran is Vice-President

has a value that is **true** or **false**.

For Amal to join the Club, it's enough that one of the statements be **true**; it's immaterial then that the other statement is **true** or **false**. Amal will join the club either if it's **true** that Bimal is President, or that Saran is Vice-President, or if both statements are **true**. The only case in which Amal will not join the Club is when both statements are **false**. Therefore, the **or** logical connective between Boolean expressions satisfies the following **truth table**:

true or true = true
true or false = true
false or true = true
false or false = false

The Or truth table

Further we are given that Bimal accepts Presidentship only if Ashok is made Vice-President and Rajni not made the Club Secretary. The statement:

Bimal accepts Presidentship

is **true** if the statement:

Ashok is made Vice President

and simultaneously the statement:

Rajni is not the Club Secretary

is **true**. For Bimal to accept Club Presidentship it's necessary that both the connected statement be **true**. It is not sufficient that Ashok is made Vice President. Nor is it enough that Rajni is not the Club Secretary. If either statement is **false**, Bimal refuses Presidentship. It is clear therefore that the **and** logical connective between Boolean values satisfies the following truth table:

true or true = true
true or false = false
false or true = false
false or false = false

The AND truth table

Again, the statement:

Rajni is not the Club Secretary

is the **negation** of:

Rajni is the Club Secretary

If the truth value of the latter is **true**, the truth value of the former is **false**. Thus **not**, which negates the truth value of the expression on which is operators, has the following truth table:

not true = false
not false = true

The NOT truth table

The problem further specifies that Rajni will not join the Club, (and so, incidentally, cannot become its Secretary) if Suman becomes a member.

That is:

if (Suman is a member)
then
(Rajni is not the Secretary);

or

if (Suman is a member)
then **not**
(Rajni is the Secretary);

or

if ((Suman is a member) = **true**)
then
((Rajni is the Secretary) = **false**).

The following additional data are provided:

Suman is a member = **true**
Ashok is Club Vice-President = **true**

thus:

Rajni is the Secretary = **not (true) = false**.

Now, the statement of the problem may be written:

if ((Ashok is Club Vice-President)

and not (Rajni is the Secretary))

then (Bimal accepts Presidentship);

i.e.

if ((true)and not (false))

then (Bimal accepts Presidentship)

i.e.

if ((true) and (true))

then (Bimal accepts Presidentship).

From the truth table for the **and** connective we find that the parenthetical condition in the last **if** evaluates to **true**. Bimal is President of the Club. Amal joins it.

You might be curious about the connection between such problems of logic and digital computers. The fact is that all of the hardware of computers and hand-held calculators—their digital circuitry—is based entirely on the laws of Boolean algebra. Every single computation that's done on a computer or calculator translates at its most basic to the evaluation of complex Boolean functions. In fact it was from the need for the exploration of the principles of computer design, that a fresh lease of life has been

given to the study of Boolean Algebra. When it was invented more than a hundred and fifty years ago, and for many years afterwards, Boolean Algebra was no more than a mathematical curiosity, and there were absolutely no practical applications of it! But here's one:

Suppose we were to design an intelligent robot that would have the capability to cross a road in the midst of traffic. At first glance this might seem to be an example of intelligent behaviour, yet the process of crossing a road is purely algorithmic, and involves the computation of several Booleans, as we'll see below.

Obviously the robot has to be equipped with cameras for input devices, that can receive traffic flow data. Its output devices would be wheels driven by motors whose speed is controlled by the CPU.

Clearly the CPU must be programmed to process the images it receives from the input. One quantity that it should be able to estimate from its input is the width of an approaching vehicle, whether it may be car, scooter-rickshaw, bus or motorcycle. This is important: it takes a longer time to cross the path of a bus, than of a motorcycle.

Apart from estimating size, the CPU should be able to gauge not only the distance of the approaching vehicle, but also its speed. This information will help the robot answer such a question as: will I be able to cross clear of the oncoming vehicle, at my maximum speed, in the time that it will take to reach me? The technology and the calculation involved are not too difficult: the robot could throw an acoustic pulse at the vehicle, and by determining the time elapsed between sending the pulse and receiving its echo compute the distance to the vehicle. By sending another pulse a short interval later, the CPU can deduce the distance travelled by the vehicle over the interval, and thus its speed.

Given its speed and distance, the robot's CPU can estimate the time an approaching vehicle will take to reach it. Then, with knowledge of the width of the approaching vehicle, and the width of the road, the CPU must evaluate Booleans such as:

if (the time the vehicle will take to reach me, say **t1**
is greater than the time I will take to
cross its path, say **t2**, at my maximum speed)
then
I'll move forward,
else
I'll wait until the road is clear.

The Boolean here is:

```
t1 > t2.
```

If it returns the value true, the robot goes ahead to cross the road, else it waits:

```
if (t1 > t2)
   I'll cross the road;
else
   I'll wait until the road is clear
```

Obviously more complex Booleans must be evaluated if the robot is to be able to cope with more than one vehicle at a time, and if vehicles are allowed to approach from either direction. Suppose there are two vehicles approaching the robot, one from the left and one from its right.

The Booleans it must evaluate before beginning to cross are:

if ((the time taken by the vehicle on my left to reach me, **t1**
is greater than the time I will take to cross its path, **t2**)

(the time taken by the vehicle on my right to reach me, **t3**
is greater than the time I will take to cross the road, **t4**))

I'll cross the full width of the road;

else

if((I can cross past the vehicle on my left)

and not

(I can cross past the vehicle on my right))

I'll cross past the vehicle on my left and pause in the
middle of the road for the vehicle on my right to pass;

else
I'll wait for the vehicle on my left to pass me by;

In a programming language these Booleans may be written:

```
if  ((t1 > t2) and (t3 > t4))
   I'll cross the road;
else
if ((t1 > t2) and not (t3 > t4))
   I'll cross over to the middle and pause
for the vehicle on my right to pass me by;
else
   I'll wait for the vehicle on my left to pass my by;
```

The next time you cross a road, contemplate with wonder on the enormous complexity of the computations your brain instinctively performs, when you use it to carry out so common an undertaking!

Realise that each **else** statement above can be associated only with the **if** physically nearest it. This is also true of analogous statements in C programs.

4.2.1 Boolean Operators

To see how Boolean relationship are written in C, suppose `x` and `y` are program variables. The following **Boolean operators** and logical connectives are available:

Operator	Usage	Meaning
!	! (Boolean)	negates Boolean expression
>	x > y	x greater than y
>=	x > = y	x greater than or equal to y
<	x < y	x less than y
<=	x < = y	x less than or equal to y
= =	x = = y	x equal to y
!=	x ! = y	x not equal to y
&&	x > y && x ! = 0	logical **and** of two Booleans
||	x > 5 || x = = y	logical **or** of two Booleans

The unary negation operator ! has a priority just below the parentheses operator: it negates the Boolean expression which follows it. Like all unary operators, it groups from right to left.

The four binary operators >, >=, and <= each have equal priority. The remaining two

operators = = and !=, which also require two operands, have equal priority, but lower than that of the first four.

&&, the logical **and** connective has a priority lower than the six relational operators above, but exceeding that of the logical **or** connective, ||. All these operators group from left to right. Each of the arithmetic operators has a higher priority than the Boolean operators; but the assignment operator has a lower precedence.

Unlike certain other programming languages like Pascal, C does not have variables of type Boolean. However, it uses two rules to get by without such variables:

(i) Every expression, **including one involving Boolean operators**, has a value.

(ii) An expression that has a non-zero value is **true**, while one that evaluates to zero is **false**.

There's a third rule that you should know, because it is often the source of puzzling exam question (though it's not of much practical use):

(iii) The value of a Boolean expression that is **true** is 1, while the value of an expression that is **false** is 0.

Thus the value of $5 > 3$ is 1, of $5 < 3$ is 0!

E1) What does the following program print?

```
/* Program 4.1; file name: unit4-prog1.c*/
#include <stdio.h>
int main()
{
    printf("%d\n", 5 > 3);
    printf("%d\n", 3 < 5);
    printf("%d %d\n", (5 > 3) && (3 < 5), (5 > 3) || (3 < 5));
    printf("%d %d\n", (1 > 0) * (1 < 0), 1 > 0 * 1 < 0);
    return (0);
}
```

Listing 1: Exercise 1.

E2) Give the output of the following program:

```
/* Program 4.2; file name:unit4-prog2.c */
#include <stdio.h>
int main()
{
    int x = 5, y = 1;
    if (x > y)
        printf("x > y\n");
    if (x < y)
        printf("x < y\n");
    if (!(x == y))
        printf("x ! = y");
    return (0);
}
```

Listing 2: Exercise 2.

E3) In Listing 1 above are the parentheses around `x == y` in the last **if** () statement necessary? Remove them, compile and execute your program again, and explain your result.

E4) In Listing 2, are the parentheses around the expression $5 > 3$ and $3 < 5$ necessary? Remove them, compile and execute your program again, and explain your result.

In Listing 3 below, the Boolean of the first **if ()**, `x > y`, is **false**; so `z` does not get the value 1. The second Boolean, `x < y`, is **true**. `z` becomes 2. The Boolean of the next **if ()**, `x == y`, is **false**, so its object statement, `x = z = 3` is not executed. Because `x` is still 2, and `y` is 3, the Boolean in the next **if ()**, `x != y`, is **true**, and `y` becomes 0. Finally, since `x` is 2 **(or true)** `!x` is **false** (or 0), so `!x == y` is **true**, and **z** gets the value 0.

```
/* Program 4.3; file name: unit4-prog3.c*/
#include <stdio.h>
int main()
{
    int x = 2, y = 3, z = 0;
    if (x > y)
        z = 1;
    if (x < y)
        z = 2;
    if (x == y)
        x = z = 3;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    if (x != y)
        y = 0;
    if (!x == y)
        z = 0;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    return (0);
}
```

Listing 3: More examples of if ()

```
/* Program 4.3 :  Output :  */

x = 2, y = 3, z = 2

x = 2, y = 0, z = 0
```

We close this section here. In the next section, we will discuss the **goto** statement that is used to control the program flow.

4.3 THE goto STATEMENT

The next program illustrates the use of the **goto** statement, which transfers control unconditionally to another place in the program that is marked by a **label**:

```
printf("Whoopee...Holidays!!!\n");
goto Goa;
: /* intervening code is skipped */
Goa: printf("See me at the beach!!\n");
```

Label names are subject to the same rules as are identifiers: they are built from the alphabetical characters, digits and the underscore, with the proviso that the first character must not be a digit. In a program a label name is followed by a colon.

The **goto** statement is held in deepest contempt by the votaries of structured programming, primarily because programs with **goto**s are extremely difficult to verify for correctness, and to debug. A program with a sprinkling of **goto**s quickly begins to resemble a plateful of noodles, and to determine the general flow of control in such a program can lead to a great deal of frustration.

Listing 4 on the next page, which uses **if ()**s and **goto**s, is concerned with a popular (and still unsolved) problem variously known as the 3N + 1 problem or the Collatz problem, after Lothar Collatz who is credited with having invented it. The problem is

easy to state: think of a whole number greater than zero; if it's odd, multiply it by 3 and add 1; if it's even, divide it by 2. Repeat these steps for the resulting number; the series of numbers generated eventually terminates in …, 4, 2, 1, no matter what number you may have begun with! For example, let's choose 7; because it's odd, we multiply it by 3 and add 1, to obtain 22; because 22 is even, we divide by 2 nd arrive at the next number, 11. The progression continues until 4, 2, 1 are reached:

7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

A proof that every number will generate a series terminating in 4, 2, 1, is still not available so the next best thing to do is to test the conjecture for as many numbers as possible: computers make this easy to do, and by the end of the eighties the Collatz conjecture had been verified for all values up to 1000000000000.

Listing 4 computes a quantity called `cycle_length`, which stores the number of iterations needed for a given **int** `input`, before the sequence terminates at 1. Its first **goto** statement transfers control right back to a label named `begin_again` if the initial `input` is less than or equal to zero.

However, if the initial input was 1, there is nothing to do, and control is transferred directly to the label `finished`, without changing the value of `cycle_length`.

Any other value of `input` is successively transformed until the last iteration yields 1, at which time the program terminates.

```
/* Program4.4; file name: unit4-prog4.c*/
#include <stdio.h>
int main()
{
    int input, cycle_length = 0;
    begin_again:
    printf("Enter a positive trail number:");
    scanf("%d", &input);
    if (input <= 0)/* only a positive input permitted */
        goto begin_again;
    iterate:
    if (input == 1)
        goto finished;
    if (input % 2 == 1) /* input was odd */
        goto odd_input;
    if (input % 2 == 0) /* input was even */
        goto even_input;
    odd_input:input = input * 3 + 1;
    cycle_length++;
    goto iterate;/* Is this statement necessary ? */
    even_input:
    input /= 2;
    cycle_length++;
    goto iterate;
    finished:
    printf("1 appeared as the terminating digit \
after %d iterations", cycle_length);
    return (0);
}
```

Listing 4: Collatz problem.

Here are some exercises for you to try.

E5) In Listing 4 above, is the statement preceding the comment:

/* Is this statement necessary> */

necessary?

E6) Follow through the steps of Listing 4 on the facing page to compute manually the value of `cycle_length` for **input** = 27. Do you agree that the **goto** should be banned?

E7) One major defect in Listing 4 on the preceding page is the presence of two **if ()** statements where one would suffice:

```
if (input % 2 == 1) etc;
```

and

```
if (input % 2 == 0) etc;
```

Obviously, **input** can be only either even or odd. If it's odd, the second **if ()** is redundant; if it's even, the second **if ()** is not required. Rewrite the program using the first **if ()** alone.

E8) Write a program to read three positive integers from the keyboard; into the **int**s **first, second** and **third**, and to print their values in descending order. You are not allowed to define any other variables in your program.

Though the use of the **goto** statement is justifiably considered reprehensible, particularly when it is used as "quick fix" to solve a problem (because it most likely will have repercussions on other parts of the program), there are instances in which a non-linear flow of control is called for, and the use of the **goto** may then be appropriate. Error handling constitutes just such a case. One of the qualities of a **robust** program is that it should check for errors in its input and be able to cope with any, by for example, prompting for fresh input or by terminating with a relevant error message. When it encounters any error, it should invoke a diagnostics routine that outputs information about the problem, and should then terminate. But this error reporting function, like all functions, can only return control to the point from which it was called, which may be nested deeply inside loops, or other program structures. The best solution may then be to transfer control via a **goto** to a label preceding statements for methodical program termination:

```
if (error)
goto report_error;
⋮
report_error:
print relevant data;
terminate program;
```

The **goto** can also be used with advantage in search procedures that may involve several nested loops, in which the value sought is determined in the innermost loop, then the statement:

```
if (found)
goto print_value:
```

in preferable to working outwards one by one through the enveloping loops. In the unit on loops we shall see just such a use of the **goto**. In the next section, we will discuss the **if ()** statement in greater detail.

4.4 THE if () STATEMENT

The **if ()** statement is perhaps the most powerful statement of any programming language. It's powerful because its Boolean argument helps the computer make choices between alternative courses of action:

if (`true`)
this statement will be executed;
if (`false`)
this statement will be skipped;
As we saw in the last program, two consecutive equal signs = = test for equality of the value of the expression which occur on either side:

```
if (input == 1)
  printf("input equals %d\n", input);
```

But this calls for caution: for probably the commonest mistake that beginning C programmers make is to write a single "equals sign", =, when they mean to test one value for equality with another:

```
if (input = 1) etc.;
```

The problem with this statement is that the parenthetical expression:

```
input = 1
```

happens to be an assignment expression, **with value 1**. Now a non-zero value inside the parentheses of an **if ()** is considered **true** (by rule (ii)), so the statement that depends on such an **if ()** is automatically executed; a side effect is that this assignment sets the value of `input` to `1`, no matter what the value that may earlier have been assigned to it! There's no question, then, of comparing `input` with `1: input` gets the value `1`!

Listing 5 brings this out clearly, and deserves to be studied with some care. It depends upon the fact that an expression has a value, which may be zero or different from zero. All non-zero values, positive or negative are **true**. Therefore, any expression, even an assignment expression such as `x = 0, x = 5`, etc. may be regarded as a Boolean in an **if ()** statement.

```
/* Program 4.5; file name: unit4-prog5.c */
#include <stdio.h>
int main()
{
    int x = 0;
    if (x = 0)
        printf("x is 0, x == 0 is true, \
but this statement will not be ouput.\n");
    if (x != 0)
        printf("x is 0, x != 0 is false,\
so this statement will not be ouput.\n");
    if (x = 5)
        printf("Can you believe it, x is indeed %d!\n", x);
    return (0);
}
```

Listing 5: A common mistake

Look at the first **if ()** statement in the program: it's Boolean argument has the value 0, since 0 is the value of the assignment expression:

```
x = 0.
```

Since a Boolean that evaluates to 0 has the value **false**, the object statement:

```
printf("x is 0, x == 0 is true, but this statement \
will not be output.\n");
```

will not be executed.

Attend now to the second **if ()** statement. It's Boolean `x != 0` is clearly **false**, since the value of `x` is 0 (observe the argument of the preceding **if ()**). The object statement is again skipped. The third **if ()** is interesting:

```
if (x = 5)
    printf("Can you believe it, x is indeed \%d!\n", x);
```

The Boolean expression:

```
x = 5
```

happens also to be an assignment expression, with value 5. Now, because a Boolean with a non-zero value it **true**, the **printf**() is executed, and prints 5 for x, which is its current value. This **if** () statement accomplishes two goals: it assigns a value to a variable; and it tests whether its Boolean argument is **true** or **false**. The simplest expression is a variable name by itself, the value of the variable being the value of the expression. Suppose that a statement must be executed only if the value of the variable is non-zero. The following **if** ()s are equivalent:

```
if(x != 0) statement;
if(x) statement;
```

Another mistake frequently encountered in C programs is the inadvertent placement of a semicolon immediately after the parenthetical Boolean in an **if** () statement as shown below:

```
if (input == 5);
   printf("The value of input is \n", input);
```

There are now two statements here instead of the single one that was actually intended: the first is an **if** () statement with a null object statement—the semicolon immediately after the Boolean; the second is the `printf()` statement which stands by itself, and which does not depend for its execution on the preceding **if** () statement; it will be executed whether or not the parenthetical Boolean is **true**. There are no syntactical errors, a program containing these statements will compile perfectly, but will execute the `printf()` no matter what value is assigned to `input`, 5 or any other. It is the null statement, the semi-colon, that is executed (or not executed) depending upon the truth value of the Boolean. The object statement of an **if** () is the **single** statement that follows it, and it may quite possibly be a null statement if the logic so warrants.

If there is a series of statements that must be executed when the Boolean of an **if** () is **true**, they must be enclosed in braces:

```
if (Boolean condition is true)
{
all the statements up to
the terminating right
brace will be executed;
}
```

A set of statements enclosed in braces is called a **compound statement**; compound statement may occur wherever the syntax allows a single statement. Compound statements help make program logic clearer; they also serve to dispense with **goto**s. The program below, a revised version of Listing 4 on page 84, has fewer **goto**s because it uses compound statements.

```
/* Program 4.6: Revised Version of Program 4.4 */
 /* file name: unit4-prog6.c */
#include <stdio.h>
int main()
{
    int input, cycle_length = 0;
    begin_again:
    printf("Enter a positive trail number.");
    scanf("%d", &input);
    if (input <= 0)
```

```
            goto begin_again;
        iterate:
        if (input == 1)
            goto finished;
        if (input % 2 == 1)
           {
            input = input * 3 + 1;
            cycle_length++;
           }
        input /= 2;
        cycle_length++;
        goto iterate;
        finished:
        printf("1 appeared as the terminating \
digit after %d iterations", cycle_length);
        return (0);
}
```

Listing 6: Collatz problem, modified program.

The goal of structured programming is to create goto-less programs. While compound object statements help to an extent in achieving this objective, it is the loop structures of C—the **for(;;)** loop, the **while()** loop and the **do-while**() loop—that are indispensable tools for accomplishing this goal. Loops will be studied in depth in the next Unit.

The object statement of an **if()** statement may be another **if()** statement. The nested **if()** is reached only if the Boolean of the covering **if()** is **true**. In the program below, the condition inside the first **if()**, `x >= 2` is **true**; its object statement will therefore be executed. As it happens, that is another **if()**, of which the Boolean is also **true**. The third **if()**, the object of the second, will consequently be reached, and since its Boolean is also **true, z** is assigned the value 4. It's a simple matter now to predict the output from the succeeding statements.

```
/* Program 4.7 *; file name: unit4-prog7.c*/
#include <stdio.h>
int main()
{
    int x = 2, y = 3, z = 100;
    if (x >= 2)
        if (y <= 3)
            if (y > x)
                z = 4;
    printf("z = %d\n", z);
    x = z = 100;
    if (x > (y = 99))
        if (y > (x = x - 2))
            z = 99;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    if (y > x)
        if (2 * z > x + y)
            {
             x = 2;
             y = 3;
             z = 4;
            }
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    if (x < 0)
        if (y > 0)
            printf("Will this statement be executed?\n");
    return (0);
}
```

Listing 7: Nested ifs.

E9) State the output of the program in Listing 7 on the facing page.

We conclude the section here. In the next section, we will discuss **if ()-else** statement.

4.5 THE if () - else STATEMENT

The **if** () **- else** statement provides for two-way branching:

```
if(condition is true)
this statement is executed;
else
this statement is executed;
```

In other words, if the Boolean in an **if () - else** statement evaluates to **true**, the object statement of the **if ()** is executed; the object statement of the **else** is skipped. Contrarily, if the Boolean evaluates to **false**, the object statement of the **if ()** is ignored, that of the **else** is executed. Where there are only two choices provided in a program the **else** statement is redundant: it's useful only when the number of choices is larger than two, as in Listing 10 on the next page below.

The following simple program illustrates the usage of the **if** () **- else** statement. It prints two numbers it scans from the keyboard in descending order.

```
/* Program 4.8; file name: unit4-prog8.c */
#include <stdio.h>
int main()
{
    int num1, num2;
    printf("Enter two numbers.\nThe first is ?");
    scanf("%d", &num1);
    printf("The second is ?");
    scanf("%d", &num2);
    printf("In descending order the numbers are:");
    if(num1 >= num2)
        printf("%d %d\n", num1, num2);
    else
        printf("%d %d\n", num2, num1);
    return (0);
}
```

Listing 8: Example program for if()-else.

Caution: Pay attention to the semicolon preceding the keyword **else**: a semicolon must terminate the object statement of an **if ()** whether or not it is followed by an else statement). An **else** statement must be preceded by, and be associated with, an **if ()** statement; it may not occur by itself. But an **if ()** statement doesn't necessarily require an **else** statement.

Listing 9 below distinguishes between people who have fun programming in C, and those who don't (inexplicably, there are some who don't).

```
/* Program 4.9; file name: unit4-prog9.c */
#include <stdio.h>
int main ()
{
    char response;
    printf("Do you like programming in C?\n");
    printf("You\'re supposed to type y, the press <CR>:");
    response = getchar();
```

```
    if(response == 'y')
        printf("Thank you, thank you, for your kindness!\n");
    if(response == 'n')
        printf("Be off with you!!\n");
    return (0);
}
```

Listing 9: "Is C programming fun?"

But what, you might ask, if the person who executes the program types in neither a `'y'`, nor an `'n'`, but a quite different and irrelevant character? Users are entitled to their foibles, you know! Then neither of the `printf()`s is executed. To handle such a situation we use again the **`if () - else`** statement so that it can cover a larger number of possibilities—**the object statement of an else may itself be an `if () - else` statement, and so on for as many times as may be needed**.

```
if(response == 'y')
   printf("Thank you, thank you, for your kindness!\n");
else
    if(response == 'n')
        printf("Be off with you!!\n");
else
   printf("We assume you don\'t want to commit yourself!");
```

```
/* Program 4.10; file name:unit4-prog10.c */
#include <stdio.h>
int main()
{
    char response;
    printf("Do you like programming in C?\n");
    printf("You\'re supposed to type y, then press <CR>:");
    response = getchar();
    if (response == 'y')
        printf("Thank you, thank you, for your kindness!\n");
    else
        if (response == 'n')
            printf("Be off with you!!\n");
    else
        printf("We assume you don\'t want to commit yourself!");
    return (0);
}
```

Listing 10: "Is C programming fun?", improved version

With these improvements Listing 10 is almost **OK** as it stands, but not quite: for what if the user types an uppercase `'Y'` when she means to say **"Yep, C's great!"** or an uppercase `'N'` when she means to say, "No thank you, C isn't quite my cup of tea!" The uppercase `'Y'` will fail the test of equality against the lowercase `'y'`, as will the uppercase `'N'` against the lowercase `'n'` (these **char**s are different because their ASCII decimal equivalents are different). Our program should be able to accept both `'Y'` and `'y'` for yes, both `'N'` and `'n'` for no. Here's one way of solving the problem:

```
/* Program 4.11; file name:unit4-prog11.c */
#include <stdio.h>
int main()
{
    char response;
    printf("Do you like programming in C?\n");
    printf("You\'re supposed to type y, then press <CR>:");
    scanf("%c", &response);
    if ((response == 'y') || (response == 'Y'))
        printf("Thank you, thank you, for your kindness!\n");
    else
```

```
        if ((response == 'n') || (response == 'N'))
            printf("Be off with you!!\n");
    else
        printf("We assume you don\'t want to commit yourself!");
    return (0);
}
```

Listing 11: "Is C programming fun?", yet another version.

The two vertical bars || represent the logical **or** operator. Since the result of an **or** is **true** if any of the connected Booleans is **true**, the program gives the appropriate output if it scans 'y' or 'Y', or 'n' or 'N' in the input.

C stops evaluating a Boolean expression as soon as it determines its truth value. Therefore in the expression:

```
response == 'y' || response == 'Y'
```

If it is **true** that `response == 'y'`, the entire expression becomes **true**, and its latter half, `response == 'Y'` is not scanned.

The **if** () - **else** statement is an aid to structured programming: it helps make programs easier to understand and debug. Consider yet another version of our program for the Collatz problem:

```
/* Program 4.12; file name:unit4-prog12.c */
#include <stdio.h>
int main()
{
    int input, cycle_length = 0;
    begin_again:
    printf("Enter a positive trail number.");
    scanf("%d", &input);
    if (input <= 0)
        goto begin_again;
    iterate:
    if (input == 1)
        goto finished;
    else
        if (input % 2 == 1)     /* input was odd */
            input = input * 3 + 1;
        else
            input /= 2;
    cycle_length++;
    goto iterate;
    finished:
    printf("1 appeared as the terminating \
digit after %d iteration(s)", cycle_length);
    return (0);
}
```

Listing 12: Program for Collatz problem, another version.

E10) In Listing 11 on the facing page, are the inner parentheses in:
`if ((response == 'y') || (response == 'Y'))` required? Why or why not?

E11) State the output of Listing 13:
Hint: An **if** () associates with the **else** physically nearest it.

```
/* Program 4.13; file name:unit4-prog13.c */
#include <stdio.h>
```

```
int main()
{
    int x = 2, y = 3, z = 100;
    if (x < 2)
        if (y >= 3)
            if (!(y > x))
                z = 4;
            else
                z = 5;
        else
            z = 6;
        else
            z = 7;
    printf("z = %d\n", z);
    x = z = 100;
    if (x > (y = 99))
        if (y > (x = x + 2))
            z = 99;
        else
            z = 101;
        else
            z = 102;
    printf("z = %d, y = %d, Z = %d\n", x, y, z);
    if (y > x)
        if (2 * z > x + y)
            {
                x = 2;
                y = 3;
                z = 4;
            }
        else
            {
                x = 4;
                y = 3;
                z = 2;
            }
        else
            x = y = z = 5;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    return (0);
}
```

Listing 13: Exercise 11.

The last program sheds some light on the importance of proper indentation. See how difficult it is to follow the logic of statements written thus:

```
if (x < 2) if (y >= 3) if (!(y > x))
z = 4; else z = 5; else z = 6, else z = 7;
```

Let's look at Listing 14 to explore the precedence relationship of the Boolean and arithmetic operators.

```
/* Program 4.14 ; file name: unit4-prog14.c*/
#include <stdio.h>
int main()
{
    int alpha = 10, beta = 5, gamma = 1, delta;
    delta = alpha > beta > gamma;
    printf("%d\n", delta);
    delta = alpha / beta == ++gamma;
    printf("%d\n", delta);
    delta = alpha || beta || gamma++ - 2;
```

```
    printf("%d\n", delta);
    delta = alpha && beta && gamma++ - 2;
    printf("%d\n", delta);
    delta = alpha / beta >= beta / gamma;
    printf("%d\n", delta);
    delta == alpha / 2 == beta;
    printf("%d\n", delta);
    delta = ++delta || (delta = 5);
    printf("%d\n", delta);
    delta = 5 || ++delta;
    printf("%d\n", delta);
    delta == delta++;
    printf("%d\n", delta);
    return (0);
}
```

Listing 14: Boolean and arithmetic operators.

In the first assignment to delta:

```
delta = alpha > beta > gamma;
```

the assignment operator has the lowest precedence. Since > groups from left to right `alpha > beta` is evaluated first, and has the value **true**, which by rule (iii) is 1. The next expression to be computed is therefore:

```
1 > gamma
```

which evaluates to **false**, and has the value 0. `delta` gets the value 0.

Consider the second assignment of `delta`:

```
delta = alpha / beta == ++ gamma;
```

The pre-incremented value of `gamma` (2) is to be compared against the quotient `alpha / beta` (also 2). Note that the quotient is available before the comparison is done. The Boolean is **true**, so `delta` is 1.

In assignment:

```
delta = alpha || beta || gamma ++ - 2;
```

it is important to realise that a Boolean is evaluated only to the extent necessary to determine its truth value. Here `alpha` is non-zero and is therefore **true**, so the entire expression is **true**; `gamma` is not post-incremented; `delta` is 1. This deserves comparison with the next assignment to `delta`, where the **OR**s are replaced by **AND**s:

```
delta = alpha && beta && gamma ++ -2;
```

Post-incrementation of `gamma` implies that its last value (2) must be used in the evaluation of `delta`. `alpha` is 10 and `beta` is 5, so

```
alpha && beta
```

is **true**, but

```
gamma ++ - 2
```

is **false**, the entire expression is **false**, and `delta` is zero. Note that `gamma` is post-incremented when the expression for `delta` involves &&s; each sub-expression must be evaluated to determine the truth value of `delta`.

In the assignment:

```
delta = alpha / beta > = beta / gamma;
```

the quotients `alpha / beta` and `beta / gamma` are computed before the inequality is tested. With the current values of `alpha, beta` and `gamma`, the inequality is **true** and `delta` is 1.

The next statement:

```
delta == alpha / 2 == beta;
```

is not an assignment statement. No rvalue is modified by it, and this may cause a warning to be issued at compile time. `delta` is still 1.

Try the following exercises now.

E12) Calculate the value that `delta` gets in the remaining assignment statement, and verify you results by executing the program.

E13) Write a program that accepts three numbers and decides:
   a) whether these can be the lengths of the sides of a triangle;
   b) if they form an equilateral, isosceles or scalene triangle
   c) if they form a right-angled triangle

E14) Write a program which accepts two numbers, and determines the following:
   a) the first is positive, negative or zero
   b) the second is positive, negative or zero
   c) the first is even or odd
   d) the second is even or odd
   e) the first is larger than the second, in absolute value
   f) if the first is exactly divisible by the second

   Hint: Divisibility of the first by the second is determinable only if the second number is different from zero.

E15) Write a program to determine whether the value of a year input to it, such as 1900, or 1996, or 2000, represents a leap year or not. Your program should prompt the user to enter a value, and it should then output one of: “Leap year” or “Not a leap year”. Hint: A leap year value is evenly divisible by 4. However, a value that is divisible by 100 is not a leap year, unless it is also divisible by 400.

E16) Give the output of the following programs:

```
/* Program 4.15; file name:unit4-prog15.c */
#include <stdio.h>
int main()
{
    int x = 10, y = 5, z = 0;
    x %= y -= z == x < y;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    x *= y += z = x == (y /= 2);
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    x = y-- && --z || (x + y - z);
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    x = y++ < z++ > ++x / 12;
    x = --x <= -y > z || x && y;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    return (0);
}
```

E17) Write a program that accepts a digit from the keyboard, and displays it in English. So, if the user types 7, the program responds with “seven”, if she types 4, the program responds with “four”, etc.

E18) Write a program that gets a character such as +, -, * or /, then scans two numbers, and then yields the result of the operation denoted by the character. In the division of one value by another, your program should behave intelligently if the denominator == 0,

---

We close this unit here. We summarise the Unit in the next section.

4.6 SUMMARY

In this unit:

1. We introduced Boolean expressions which perform certain tests for taking decisions on the flow of control in the program. They are built up from Boolean operators like ||, && and comparison operators like ==.Boolean expressions are assigned the value 1 when they are true and 0 when they are false.
2. We discussed the **goto** statement. While this has to be avoided for the sake of clarity, it could be useful some times.
3. We discussed the **if ()** statement whose usage is as follows:

   if()

   if(*Expression*)
   statement or compound statement;

   If the value of the *Expression* is non-zero, the instructions in the statement or compound statement following it is executed; otherwise it is skipped. We can use any valid C expression here, not necessarily a Boolean expression.
4. We discussed an extension of **if ()**, the **if ()-else** whose usage is as follows:

   if()-else

   if(*Expression*)
   statement or compound statement;
   else
   statement or compound statement;

If the value of the expression is non-zero, the instructions in the statement or compound statement following it are executed. Otherwise, the instructions in the statement or compound statement after the **else** are executed. As in the case of **if ()** any valid C expression can be used, not necessarily a Boolean expression.

4.7 SOLUTIONS/ANSWERS

E1)
```
1
1
1 1
0 0
```

E2)
```
x > y
x !  = y
```

E3) No, because the unary operator ! has a higher precedence than the binary operator ==. Hence !5 will be calculated and the value is 0. So, the value of !x == y will be true and x != y will be printed. However, if y is different from 0, you will get entirely different results! Try it now! Change the value of y to 1, remove the brackets around x != y and see what you get!

E4) No, since && has lower priority than the < or >, the expression is evaluated we want.

E5) No, if the statement is not there, the program will divide input by 2 and increase the cycle length 1 also. So, the program will correctly perform the required operations and do the book keeping also correctly without the iterate.

E7)
```
/* Program4.4 :File unit4-ans-ex7.c*/
#include <stdio.h>
int main()
{
    int input, cycle_length = 0;
    begin_again:
    printf("Enter a positive trail number:");
    scanf("%d", &input);
    if (input <= 0)/* only a positive input permitted */
        goto begin_again;
    iterate:
    if (input == 1)
        goto finished;
    if (input % 2 == 1) /* input was odd */
        goto odd_input;
    input /= 2;
    cycle_length++;
    goto iterate;
    odd_input:input = input * 3 + 1;
    cycle_length++;
    goto iterate;
    finished:
    printf("1 appeared as the terminating digit \
after %d iterations", cycle_length);
    return (0);
}
```

E8)
```
/*File name: unit4-ans-ex8.c*/
#include<stdio.h>
int main()
{
    int first, second, third;
    printf("Enter the first number..\n");
    scanf("%d", &first);
    printf("Enter the second number..\n");
    scanf("%d", &second);
    printf("Enter the third number..\n");
    scanf("%d", &third);
    /*If all the numbers are equal
    print them*/
    if(first == second && second == third)
        goto caseb;
    if(first <= second)
        goto casea;
    /*Now, second < first;
    If third <= second,
    the order is first >= second >= third*/
    if(third<=second)
        printf("%d, %d, %d", first, second, third);
```

```
    /* If first <= third, the order is
    third>=first>=second*/
    if(first <= third)
        printf("%d, %d, %d", third, first,second);
    /*The only possibility now is
    second<=third<=first*/
    if(second <= third && third <= first)
    printf("%d, %d, %d", first, third, second);
    goto finished;
    caseb:
    /*All the numbers are equal; print the numbers
    and end the program*/
    printf("%d, %d, %d", first, second, third);
    goto finished;
    casea:
    /* Now, first <= second;If third <= first,
    the order is third, first, second*/
    if(third <= first)
        printf("%d, %d, %d", second, first, third);
    /*If second is less than third, the order is
    third, second, first*/
    if(second <= third)
        printf("%d, %d, %d", third, second,first);
    /*The only possiblity is
    first, third, second*/
    if(first <= third && third<=second)
        printf("%d, %d, %d",second,third,first);
    finished:
    return 0;
}
```

E9) **z = 4**

x = 98, y = 99, z = 99

x = 2, y = 3, z = 4

E10) No, they are not required. Because == has higher priority than ||, the expression will be evaluated correctly without the parentheses. The parentheses improve the readability of the programme.

E11) **z = 7**

z = 102, y = 99, Z = 101

x = 5, y = 5, z = 5

E12) The value of `delta` in all the statements except the last one is 1 because `delta` is assigned the result of **OR** operator applied to 2 non-zero quantities. In the statement `delta == delta++` no assignment is made to `delta` but it is post-incremented and gets the value 2.

E13)
```
#include <stdio.h>
int main()
{
    int a, b, c;
    printf("Enter Side 1 ..");
    scanf("%d",&a);
    printf("Enter Side 2 ..");
    scanf("%d",&b);
    printf("Enter Side 3 ..");
    scanf("%d",&c);
    if( (a+b <= c) || (a+c <= b) || (b+c <= a))
        {
```

```
            printf("%d, %d and %d do not form a triangle",a,b,c);
            goto finished;
           }
    else
        printf("%d, %d and %d do form a triangle.\n",a,b,c);
    if((a == b) || (b == c))
        {
         printf("%d, %d and %d form an \
isoceles triangle.\n",a,b,c);
         if( (a ==b )&& (b == c))
             printf("%d, %d and %d actually form an \
equilateral triangle.\n", a,b,c);
        }
    else
        printf("%d, %d and %d form a scalene \
triangle.\n",a,b,c);
    /*Apply Pythagoras theorem.*/
    if( (a*a + b*b == c*c) || (a*a + c*c == b*b)
        || (b*b + c*c == a*a))
        printf("%d, %d and %d form a right angled \
triangle.",a,b,c);
    else
        printf("%d, %d and %d do not form a right \
angled triangle.",a,b,c);
    finished:
    return (0);
}
```

E14) The following program checks which of the numbers are positive, which are negative and which are zero. Complete the program to carry out the remaining checks.

```
#include <stdio.h>
int main ()
{
    int a, b;
    printf("Enter the first number ...\n");
    scanf("%d",&a);
    printf("Enter the second number ...\n");
    scanf("%d",&b);
    if ( a*b == 0)
        {
         if (a == 0)
             {
              printf("First number is zero.\n");
              if (b > 0)
                  printf("Second number is positive");
              else
                  if(b < 0)
                      printf("Second number is negative\n");
              }
         else
             {
              printf("The second number is zero.\n");
              if (a > 0)
                  printf("The first number is positive.\n");
              else
                  printf("The first number is negative.\n");
             }
        }
    else
        {
         if( a*b > 0)
```

```
                {
                 if ( a > 0)
                     printf("Both the numbers are positive");
                 else
                     printf("Both the numbers are negative.\n");
                }
            else
            /* a*b < 0 is the only possibility now.*/
                {
                 if ( a < 0)
                     printf("First number is negative, \
    second number is positive.\n");
                 else
                     printf("First number is positive, \
    second number is negative.\n");
                }
            }
        return (0);
    }
```

E15)
```
    #include <stdio.h>
    int main()
    {
        int year;
        printf("Enter the year...\n");
        scanf("%d",&year);
        if ( year % 4 == 0)
            if (year % 100 == 0)
                if( year % 400 !=0)
                    printf("The year %d is not a leap year.",
    year);
                else
                    printf("The year %d is a leap year.",year);
            else
                printf("The year %d is a leap year.",year);
        else
            printf("The year %d is not a leap year.",year);
        return (0);
    }
```

E16) Output of program:

```
x = 2, y = 4, z = 0
x = 6, y = 3, z = 1
x = 1, y = 2, z = 0
x = 1, y = 3, z = 1
```

E17) We have given a program skeleton that reads out 0 and 1. Complete the rest of the program on your own.

```
#include <stdio.h>
int main()
{
    char number;
    printf("Enter a number from 0 to 9 ...\n");
    scanf("%c",&number);
    if (number == '0')
        printf("You entered zero.");
    if (number == '1')
        printf("You entered one.");
    /* Fill in the skeleton.*/
    return (0);
}
```

E18) We have given a program skeleton that performs division. Complete the rest of the program on your own.

```
#include <stdio.h>
int main()
{
    char operator;
    float a, b;
        printf("Type a simple expression like 3*4 to \
evaluate:\n");
        printf("Do not include spaces.\n");
        scanf("%f%c%f",&a,&operator,&b);
    if(operator == '/')
            {
                 if( b == 0)
                    printf("Division by 0 not possible.");
             else
                    printf("The answer is %f",a/b);
            }
    return (0);
}
```

UNIT 5 CONTROL STRUCTURES - I

Structure **Page No.**

5.1 INTRODUCTION

The rationale of structured programming is to create **goto**-less programs. While compound object statements help to an extent in achieving this objective, it is the loop structures of C—the **for (;;)** loop, the **while()** loop and the **do - while()** loop—that are indispensable tools for accomplishing this goal. Loops are required wherever a set of statements must be executed a number of times. In this Unit, we will introduce to loops. In Sec. 5.2, we will discuss the **while()** and **do-while()** loops. We will also introduce you the the **switch-case-default** statement in Sec. 5.6. In Sec. 5.4, we will discuss how to transfer control from within loops. In Sec. 5.3, we will introduce you to the comma operator.

Objectives

After studying this unit, you should be able to

- write programs using the **while ()** and **do - while ()** loops;
- explain the use of comma operator;
- use the ternary of **if - then - else** operator; and
- use the **switch - case - default** decision structure for multiple-way branchings.

5.2 THE while () AND do - while () LOOPS

Let us begin our discussion of loops by looking at a common task you may have performed many times. Suppose you want look up a word in the dictionary.For example, you are looking up the word **ringent**. It's unlikely that you will be able to open the dictionary at the right page and locate the word in its line the first time; so you open it towards the middle, where let's assume you find the word **nexus**. Since **n** comes before **r**, the latter half of the book must now be searched. Thus the process is repeated: you open the book towards the middle of the second half, at approximately the three-quarters point, where again assume you find **scrub**. **s** comes after **r**, so **ringent** must occur, if at all it does, between the pages which contain **nexus** and **scrub**; you therefore bisect this interval, and suppose you now find the dictionary open at the page containing **puddle**. **p** comes before **r**, so **ringent** must be between **puddle** and **scrub**. You therefore divide this interval, and thus continue to bisect the ensuing sub-intervals, until you find the word, or discover that it's not defined in the dictionary. Note that, beginning with the entire dictionary as the original search interval, you have executed the interval creation and search instruction repetitively; in computer jargon, you've executed a loop. Here's the algorithm:

```
interval = entire dictionary;
while (word has not been found)
    {
     bisect interval;
     of the two intervals created, determine likely_interval;
     interval = likely_interval;
     if (interval > 0)
         continue;
     else
     break out from the loop;
    }
```

This search procedure is appropriately enough called the **bisection method**; note that it can work only if the entries are sorted in ascending order, such as words in a dictionary, or names in a telephone directory. Each execution of the loop instruction—the statements inside the curly braces—is an **iteration**, and is performed only if the Boolean at the beginning of the loop is **true**: that is, if the word is still not found. Like Pascal, C has three loops, two of which, the **while**() and the **do-while**() are introduced in this unit. In the **for** (;;)and the **while**() loop, the Boolean is evaluated **before** the loop is entered; the loop is skipped if the Boolean is **false**. In the **do - while**() loop, the Boolean sits at the **end** of the loop statements, which are executed anyway the first time around even before the Boolean has been examined. If the Boolean is then found to be **true**, the loop is iterated, else not. So a **do - while**() loop differs from the other two in this important particular: it is invariably executed at least once.

The **break** statement forces an immediate exit from the loop. The **continue** statement forces control to be transferred back to a re-evaluation of the Boolean controlling the loop. If it is **true**, the loop is again executed, else the loop is exited. The **break** and the **continue** statements may be used in each of the loop structures of **C**; the **continue** statement cannot be used outside a loop; the only other place, besides loops, where the **break** statement may be used is in the **switch - case - default** statement, discussed later in this unit; **break** and **continue** are keywords, and may not be used in any contexts other than those stated.

Sometimes it is desirable to be able to transfer control to one of several possible execution paths in a program, to the exclusion of all the others. In driving your car in New Delhi, you may find yourself a a roundabout of several roads radiating outwards, of which only one will (optimally) take you where you want to go. You have to decide from the possibilities available, and choose the appropriate path. In doing so, you will be executing the analogue of a **C** decision structure called the **switch - case - default** statement; this directs the flow of control to one of the several **case**s listed in the statement, depending upon the value taken by an integer variable, called the **switch** variable. Just as a driver at a roundabout chooses one and only one radial road, control flows in a **switch** statement along a unique path (among several that may be listed) determined by the **switch** variable.

This Unit introduces loops and the **switch - case - default** decision structure.

For a quick look at the **while()** and **do - while ()** loops, lets rewrite Program 4.12—our program for the Collatz problem—without using the **goto** statement. This will clear the way for us to see how Booleans control loops. And you will learn how loops make the **goto** unnecessary. For your convenience, we reproduce that program (with its many faults) below:

```
/* Program 4.12; file name:unit4-prog12.c */
#include <stdio.h>
int main()
{
    int input, cycle_length = 0;
```

```
    begin_again:
    printf("Enter a positive trail number.");
    scanf("%d", &input);
    if (input <= 0)
        goto begin_again;
    iterate:
    if (input == 1)
        goto finished;
    else
        if (input % 2 == 1)     /* input was odd */
            input = input * 3 + 1;
        else
            input /= 2;
    cycle_length++;
    goto iterate;
    finished:
    printf("1 appeared as the terminating \
digit after %d iteration(s)", cycle_length);
    return (0);
}
```

Listing 1: Collatz problem again

Note that the first **goto** in Program 4.12 transfers control over the over again to the label `begin_again` while the value scanned for the variable named `input` is negative or zero. But if the value entered for `input` is positive, the preceding **if()** statement ensures that the **goto** is not executed.

```
begin_again:
    printf("Enter a positive trail number.");
    scanf("%d", &input);
    if  (input < = 0)
        goto begin_again;
```

In effect we have the following situation:

```
do
    {
     printf("Enter a positive trail number.");
     scanf("%d", &input);
    }
while(input < = 0)
```

In the **do - while**() loop, the parentheses following the keyword **while** contain a Boolean expression. For so long as the Boolean expression is **true**, (yet at least once) the statement sandwiched between the keyword **do** and the **while()** (which may be a compound statement if need be), is repeatedly executed. But if, at the end of any iteration, the Boolean is found to be **false**, the loop is exited. Because the truth or falsity of the Boolean is established in the last statement of the loop, and therefore only after the loop has been entered, the intervening statement is necessarily executed at least once. In the present instance, if the value scanned for `input` exceeds zero, the loop is exited after the first execution of its statements. But if the `input` is less than or equal to zero, the terminating Boolean remains **true**, and the program asks for another value for `input`. Once acceptable value for `input` must be got; if it's right the first time, the loop is exited immediately; if it's not, the loop is executed over and over until a usable value for `input` has been entered. In using the **do - while()** loop, we've rid ourselves of the first **goto**. See Listing 2 on the following page.

Moral: Use the **do - while()** loop wherever a statement (or a set of statements, in which case they must be enclosed in braces) is to be executed at least once, and possibly oftener.

Furthermore, in Program 4.12, the statements between the labels `iterate` and `finished` are repeatedly executed as long as the value of `input` is different from 1.

```
iterate:
if (input ==1)
    goto finished;
else
    if (input % 2== 1)
        input = input * 3+1;
    else
        input /=2;
cycle_length ++;
goto iterate;
finished :
```

This is just the sort of situation where the **while()** loop is appropriate:

```
while (/* Boolean is true, i.e. */ input !=1)
    {
     execute these statements;
    }
```

To start with, the loop is entered only if the Boolean is **true**, i.e. if `input !=1`. If the Boolean is **false**, control skips the loop. That takes care of the statement:

```
if (input == 1)
    goto finished;
```

The loop is executed over and over again as long as the Boolean remains **true**—its truth value is determined before each subsequent execution of the loop. Thus, if the value entered for `input` is 1, the loop will not be entered. (The Collatz conjecture is in this case **true**; no further processing is necessary.) But if `input` is different from 1, the loop will be executed repeatedly, until it becomes 1 (which, we believe, will happen at sometime or the other for arbitrary positive integers treated to the Collatz algorithm).

```
while (input !=1)
   {
    if (input % 2 == 1)  /* input was odd */
        input = input * 3+1;
    else
        input /=2;
        cycle_length ++;
    }
```

Note that the index of the loop—`input`—is modified inside the loop. The loop is executed repeatedly as long as the value of `input` is different from 1; that takes care of the last **goto** of Program 4.12; and control exits from the loop as soon as `input` becomes 1; this makes the second **goto** statement in the program superfluous.

Because Listing 2 uses loops instead of **goto**s, see how much more readable and elegant it has become than our earlier version.

```
/* Program 5.1; file name:unit5-prog1.c */
# include <stdio.h>
int main()
{   int input, cycle_length = 0;
    do
        {
         printf("Enter a positive trial number:");
         scanf("%d", &input);
        }
    while (input <= 0);/*End of do*/
    while (input != 1)
        {
```

```
        if (input % 2 == 1)    /* input was odd */
            input = input * 3 + 1;
        else
                input /= 2;
                cycle_length++;
       }
    printf("1 appeared as the terminating digit \
after %d iteration(s)", cycle_length);
    return (0);
}
```

Listing 2: Collatz problem using while() loop.

For another example of the **do - while** and **while()** loops, let's write a program to find the sum of digits of a number. The algorithm is straightforward: while `number` is different from zero extract its rightmost digit using the % operator with 10 as divisor, add it to a variable `sum` that is initially zero, then discard the rightmost digit of `number` by dividing it by 10, replacing `number` by this quotient; and repeat these steps for the new value of `number`, until it reduces to zero, atwhich time the Boolean controlling the **while()** becomes **false**. See Program 5.2 below:

```
/* Program 5.2; File name: unit5-prog2.c */
#include <stdio.h>
int main ()
{
    int number, sum_of_digits = 0;
    printf ("Enter a ");
    do
        {
         printf("positive number only:");
         scanf("%d", &number);
        }
    while (number <= 0);
    while (number)
        {
         sum_of_digits += number % 10;
         number /= 10;
        }
    printf("The sum of digits is %d", sum_of_digits);
    return (0);
}
```

Listing 3: Finding the sum of the digits.

Note that braces are required in a **do - while()** loop when there is more than one statement sandwiched between the keywords **do** and **while()**;

Continuing further with the last example, let's now tackle a more challenging problem: find a four digit number such that its value is increased by 75% when its rightmost digit is removed and placed before its leftmost digit.

In the previous example the value of the input variable `number` was successively reduced by a factor of 10 in each iteration, until it eventually became zero, making the Boolean contained in the **while()** loop **false**. But this time we'll need to store `number` unchanged, in order to effect the comparison with the value created after the transposition, called `new_number` in Listing 4 on the following page below.

Assume the rightmost digit is stored in `rdigit`. Then `new_number` must be `1000 * rdigit + number / 10`. Now it is quite possible that there may be no numbers at all satisfying the property sought. The **int** variable `sentinel`, initialised to 0, records the number of successes obtained. If there were none, the program reports this on exit from the **while()** loop.

```
/* Program 5.3; File name:unit5-prog3.c */
#include <stdio.h>
int main()
{   int number = 1000, new_number, rdigit, sentinel = 0;
    while (number < 9999)
    {
     rdigit = number % 10;
     new_number = rdigit * 1000 + number / 10;
     if (4 * new_number == 7 * number)
         {
          sentinel++;
          printf("%d has the required property.\n", number);
         }
     number++;
    }
    if (sentinel == 0)
        printf("There were no 4-digit numbers \
with the property.\n");
    return (0);
}
```

Listing 4: Use of sentinels.

Here is the output of the program:

```
/* Program 5.3 - output */

1212 has the required property.
2424 has the required property.
3636 has the required property.
4848 has the required property.
```

Fibonacci
1170–1250

For another example of the **while()** loop, let's look at Listing 5 below to generate the Fibonacci numbers, introduced in a foregoing Unit. This famous sequence of numbers is named after a pre-Renaissance Italian mathematician named Leonardo Fibonacci (which quite literally means "son of Bonacci"), who first discussed them in his book Liber Abaci ("Book about the Abacus") written in 1202. Fibonacci posed the question: How many pairs of rabbits can be produced from a single pair in one year, if every month each pair gives rise to one new pair, and new pairs begin to produce young two months after their own birth. A little thought will convince you that, provided no mishaps such as famine or predatory attacks befall them, and that each pair born comprises of a male and a female, the following numbers of rabbit pairs will be born in each succeeding month:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

You can see that from the third month onwards, the number of pairs produced in any month is the sum of the numbers produced in the two preceding months. With this knowledge it's a fairly simple matter to write a program that will give the number of rabbit pairs produced in any month:

```
/* Program 5.4; File name: unit5-prog4.c */
#include <stdio.h>
int main()
{   int fib1, fib2, pairs, loop_index = 3, month;
        printf("Rabbit pairs produced in which month?...: ");
    scanf("%d", &month);
    if (month == 1 || month == 2)
        pairs = 1;
    else
        {
         fib1 = fib2 = 1;
         while (loop_index++ <= month)
```

```
            {
             pairs = fib1 + fib2;
             fib1 = fib2;
             fib2 = pairs;
            }/*End while*/
        }/*End else*/
    printf("The number of pairs produced in \
month %d is %d.\n", month,pairs);
    return (0);
}
```

Listing 5: Fibonacci Sequence.

To quickly recapitulate the concepts we've discussed so far, let's work through Listing 6 below:

```
/* Program 5.5; File name:unit5-prog5.c */
#include <stdio.h>
int main ()
{   int a, b, c;
    a = b = 5;
    while (b < 20)
        {
         c = ++ a;
         b += ++ c;
        }
    printf("a = %d. b= %d, c = %d\n", a, b, c);
    a = b = 5;
    do
        {
         c = ++ a;
         b += ++ c;
        }
    while (b <= 30 && c <= 12);/*End do*/
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    a = b = 0;
    if (++ a)
        if (-b)
            while ((c = a + b) < 5)
                {
                 ++ a;
                 ++ b;
                }
    printf("a = %d, b = %d, C = %d\n", a, b, c);
    return (0);
}
```

Listing 6: A recapitulation.

In the first **while()**, the Boolean `b < 20` is initially **true**, the loop is entered, its first statement assigns 6 to `c`, while the second sets `c` at 7 and `b` at 12. The Boolean controlling the **while()** remains **true**, so the loop is entered again. `c` is reset to 7, the next statement makes it 8, and assigns 20 to `b`. `b < 20` now becomes **false**; the values printed for `a, b` and `c` are respectively 7, 20 and 8.

The statements of the next loop, the **do - while()**, are executed unconditionally the first time around. Its first statement sets `a` and `c` at 6 each, while its second increments `c` to 7 and assigns 12 to `b`. These values for `b` and `c` are such that the loop's Boolean remains **true**, so the loop is re-entered. `a` is incremented and `c` is reset to 7, it then becomes 8, and `b` becomes 20. The Boolean continues to remain **true**, the loop is therefore once again executed. `a` and `c` are again reset at 8 in the first statement of the loop, then `c` becomes 9, `b` becomes 29, and the loop is re-entered. `c` and `a` are set at 9, the next statement resets `c` to 10 and `b` to 39, at which time the Boolean becomes **false**,

and the loop is exited, with a, b and c equalling 9, 39, and 10 respectively.

Proceeding further, we find:

```
a = b = 0;
if ( ++ a)
    if (- b)
        while ((c = a + b ) < 5)
            {
             ++ a;
             ++ b;
            }
```

The Booleans with each of the **if**s() are **true**, since they have values different from 0; so the **while()** can be reached. At this time a is 1, b is -1, and c is 0. Because c = 0, (less than 5), the **while()** loop is entered; a is incremented to 2, b to 0, and c is assigned the value 2, so that the Boolean controlling the **while()** remains **true**. Then a becomes 3, b becomes 1, c is assigned 4, and the loop is reentered. Next a becomes 4, b becomes 2, c becomes 6 and the loop is exited. Try the following exercise to check your understanding of the **while** and **do-while** loops.

E1) For a prime p, the ring $\frac{\mathbb{Z}}{p\mathbb{Z}}$ is a finite field of p elements and the non-zero elements of the field form a cyclic group of order $p-1$. Write a program that reads in a prime p and a natural number a less than p and finds the order of a in $\frac{\mathbb{Z}}{p\mathbb{Z}}$.

We close this section here. In the next section, we will discuss the comma operator.

5.3 THE COMMA OPERATOR

We'll digress for a moment from our study of loops to introduce a new C operator, used most frequently in loop control expressions: this is the humble comma. It, too, belongs to the prestigious club of operators in C; however, it suffers the ignominy of having the lowest priority of all of the members of that club. Every other operator gets precedence over the comma operator. Nonetheless, it serves a useful purpose. **When expressions in a C statements are separated by commas, it is guaranteed that they will be evaluated in left to right order.** As an expression is evaluated, its value is discarded, and the next expression is computed. The value of the expression is the last value that was computed.

Consider the statement:

x = y = 3, z = x, w = z;

The expression:

x = y = 3

will be evaluated first, and will impart the value 3 to y and to x (in that order). The expression:

z = x

will be evaluated next, and since x is 3 by this time, z is 3, too. Finally w gets the value 3.

Commas that are used to separate variable names in a declaratory or defining statement, or to separate a list of arguments (which may be expressions) in a function's invocation (such as the **printf()**), do not constitute examples of the comma operator, and it is not guaranteed that the listed expressions will be evaluated in left to right order.

Program in Listing 7 illustrates the role of the comma as an operator. The output of each program is appended. Consider first Listing 7.

```
/* Program 5.6; file name: unit5-prog6.c */
# include <stdio.h>
int main ()
{
    int x, y , z;
    x = 1, 2, 3;
    printf("x = %d\n", x);
    x = (1, 2, 3);
    printf("x = %d\n", x);
    z = (x = 4, 5, 6) / (x = 2, y = 3, z = 4);
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    return (0);
}
```

Listing 7: An example of role of comma operator.

In the expression:

x = 1, 2, 3

the comma operator guarantees that the first expression to be evaluated is the leftmost expression, which assigns the value 1 to x. The assignment expression is then discarded and the next expression, 2, is evaluated in which no operation is performed. It is similarly discarded. The value of the last expression is 3, and that is the value of the entire expression. The value printed in the first line of output is the current value of x.

In the expression:

x = (1, 2, 3)

the parentheses ensure that the expression:

1, 2, 3

is evaluated first, which, by virtue of the comma operator, gets the value 3. This is the value that is assigned to `x`, the value written in the second line of output.

In the last assignment statement in the program:

z = (x = 4, 5, 6) / (x = 2, y = 3, z = 4);

the numerator and denominator expressions get the values 6 and 4 respectively. `z` is 1.

```
Program 5.6:  Output x = 1
x = 3
x = 2, y = 3, z = 1
```

The fact that `x` gets the value 4 in the third line of output is intriguing. It implies that the denominator was evaluated **after** the numerator, in which `x` was set equal to 2. However, this order is purely compiler dependent. This output was obtained after compiling the program using gcc. The order may be different in a different compiler, i.e. the numerator may be evaluated after the denominator. Effects which arise from a compiler dependent (and therefore non-unique) sequence of execution of operations are called side effects. They're detrimental both to program robustness and portability, and should be avoided.

Commas in the argument list of a declaratory statement, or an invoked function such as a **printf()**, play the role of separator rather than operator, and it is not guaranteed that the separated expressions will be evaluated in left to right order. Listing 8 below was executed in VAX C and on a PC with ANSI C. Note the differences in the outputs.

```
/* Program 5.7; file name = unit5-prog8.c */
```

```
#include <stdio.h>
int main()
{   int x = 1, y = (x -= 5), z = (x = 7) - (x = 3);
    printf(" x = %d, y = %d, z = %d\n", x, y, z);
    printf(" x = %d, y = %d, z = %d\n",
x = ++y, y = ++x, z = ++x + ++y);
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    return (0);
}
```

Listing 8: Non-unique order of evaluation in printf().

```
Program 5.7:  Output from VAX C
x = 3, y = -4, z = 0
x = 6, y = 5, z = 1
x = 6, y = 6, z = 1
Program 5.7:  Output from a PC based ANSI C compiler
x = 3, y = -4, z = 4
x = 6, y = 5, z = 1
x = 6, y = 6, z = 1
```

Moral: Avoid program statements in which the order of evaluation of embedded expressions is undefined in the languages.

E2) Modify Program 5.4 so that it continues to give the correct output after making the following changes:
   (i) Replace the **while()** loop by a **do - while()** loop.
   (ii) Replace `loop_index ++` by `++ loop_index`.

Often, we may want to terminate a loop in between or to transfer the control outside the loop. We will see how to do this in the next section.

5.4 THE TRANSFER OF CONTROL FROM WITHIN LOOPS

There are several ways by which the execution of a loop may be terminated. There is of course the **goto** statement but its use is frowned upon, though it can provide a useful escape route from deeply nested loops (loops within loops within loops...); more appropriate are the **break** and the **continue** statements, and the **return** statement; this last however can be used only inside a function other than **main()**, and we shall defer its discussion to a subsequent unit.

The **break** statement transfers control to the statement following the **do - while(), while(), or for(;;)** loop body. The **continue** statement transfers control to the bottom of the loops and thence to a re-evaluation of the loop condition. In the **do - while() and while()** loops, the **continue** statement causes the Boolean within the **while()** to be re-evaluated; loop execution is continued if the Boolean there remains **true**. The **continue** statement cannot be used outside loops.

Listing 9 illustrates how the **break** statement enables control to be transferred outside a **while()** loop. It computes the month in which, beginning at month 2, rabbits reproducing according to the Fibonacci formula will exceed a limit scanned from the keyboard.

```
/* Program 5.10; file name:unit5-prog10*/
#include <stdio.h>
```

```
int main ()
    {
    int fib1 = 1, fib2= 1, fib3, limit, month = 2;
    do
        {
         printf("Upper limit for fibonacci\'s \
rabbits? (must exceed 1):");
         scanf("%d", &limit);
        }
    while (limit <= 1);
    while (fib3 = fib1 + fib2)
        {
         month ++;
         if (fib3 >= limit)
             break;
         fib1 = fib2;
         fib2 = fib3;
        }
    printf("Limit is reached or exceeded \
in month: %d.\n", month);
    return (0);
}
```

Listing 9: The break statement.

The **continue** statement transfers control back to a re-evaluation of the loop condition for **while()** and a **do - while()** loops. For a simple example of the **continue** statement, let's look at Listing 10, which finds out whether a number input to it is a perfect square (that is, if it has an integer square root). The logic is straightforward:

```
Assign 1 to loop_index
if (loop_index * loop_index < input)
    increment loop_index and continue;
else
    break;

/* Program 5.11; file name: unit5-prog11.c */
#include <stdio.h>
int main ()
        {
    int input, loop_index = 1;
    do
        {
         printf("Enter a number greater than 1...");
         scanf("%d", &input);
        }
    while (input <= 1);
    while (loop_index ++)
        if (loop_index * loop_index < input)
            continue;
        else
            break;
    if (loop_index * loop_index == input)
        printf("%d is a perfect square; \
square root = %d.\n", input, loop_index);
    else
        printf("The square root of %d \
lies between %d and %d.\n",
input, loop_index - 1, loop_index);
    return (0);
}
```

Listing 10: Example of break statement.

The **continue** statement causes `loop_index` to be incremented for as long as `loop_index * loop_index < input`; the square root of **input** has been neither reached nor exceeded; but when either of these events happens, the **break** statement forces control out of the loop. The next statement prints the square root if there is an exact one, or else the integers between which it lies.

C loops may be nested one inside the other, to any depth. The only condition is that any included loops must be entirely enveloped within the surrounding loop. Processing inside nested loops may give rise to a situation when the **goto** statement may justifiably be used to transfer control from the innermost loop to outside all of the surrounding loops, as soon as a desired condition is found, instead of letting control fall through all of the intervening **break** statements. While the **goto** must be avoided to the extent possible, one must not make a fetish of avoiding it. Here's a problem for which a **goto** transferring control out of nested loops is appropriate:

Three school girls walking along see a car hit a cyclist. The driver drives off without stopping. They report the matter to the police like this: the first girl says "I couldn't see the entire number of the car, but I remember that it was a four-digit number and its first two digits were divisible by 11." The second girl says, "I could see only the last two digits of the car's number, and they too formed a number divisible 11." The third girl said, "I remember nothing about the number, except that it was a perfect square." Write a C Program to help the police trace the car.

Clearly, the number we seek is a square number of the form **aabb**, where **a** and **b** will range from 1 through 9. (Why shouldn't **b** range from 0 through 9?) Listing 11 solves the problem.

```
/* Program 5.12; file name:unit5-prog12.c */
#include <stdio.h>
int main ()
        {
     int a = 1, b, sqroot;
     while (a <= 9)
         {
          b = 1;
          while (b <= 9)
              {
               sqroot = 32;
               while (sqroot ++)
                   {
                    if (sqroot * sqroot <
                        1100 * a + 11 * b)
                        continue;
                    else
                        if(sqroot * sqroot ==
                            1100 * a + 11 * b)
                            goto writeanswer;
                        else
                            break;
                   }
               b ++;
              }
          a ++;
         }
    writeanswer: printf("Car\'s number is : \
%d\n", 1100 * a + 11 * b);
    return (0);
}
```

Listing 11: Use of break statement.

E3) How many terms of the series:

$$1 * 1 + 2 * 2 + 3 * 3 + ... + n * n$$

must be summed before the total exceeds 25,000?

E4) Referring to Listing 11 on the preceding page, why should the variable `sqroot` begin at 33 in the innermost **while()** loop?

E5) Rewrite the program in Listing 11 on the facing page without using a **goto** statement.

We close this section here. In the next section, we will discuss **if-then-else** operator.

5.5 THE (TERNARY) if() - then - else OPERATOR

The **if - then - else** operator has three operands, the first of which is a Boolean expression, and the second and third are expressions that compute to a value:
first_exp ? second_exp : third_exp
The three expressions are separated by a question mark and a colon as indicated. The operator returns the value of `second_exp if first_exp` is **true**, else it returns the value of `third_exp`. Hence its name: the **it - then - else** operator. In other words, the ternary operator returns a value according to the formula:
Boolean_expression ? this_value_if_true : else_this
Semantically, therefore, the statement which assigns one of two values to a variable `x` via the ternary operator:
x = first_exp ? second_exp : third_exp;
is equivalent to:

```
if (first\_exp)
    x = second\_exp;
else
    x = third\_exp;
```

Often there's little to choose between the two alternative mechanisms of assigning a value to `x`; but the ternary operator makes for more concise and elegant, (though at times obfuscating) code.

The ternary operator has a priority just above the assignment operator, and it groups from right to left. *If* **second_exp** *and* **third_exp** *are expressions of different types, the value returned by the ternary operator has the wider type, irrespective of whether first_exp was true or false*. For example:

```
x = (5 > 3) ? (int) 2. 3 : (float) 5;
```

will return to `x` the value 2.0, rather than 2.

E6) Give the output of the following program:

```
/* Program 5.13; file name:unit5-prog13.c */
#include <stdio.h>
int main()
{   int i = 1;
    if (i / (5 > 3) ? (int) 2.3 : (float) 5)
        printf("Will this line be printed?\n");
    else
        printf("Or will this line be printed?\n");
    return (0);
}
```

E7) Would the output of the program above be different if the cast operator (**float**) was removed from the third operand of the ternary operator?

E8) Observer that the following statement assigns the lesser of two values `val_1` and `val_2` to a variable named lesser:

```
lesser = val_1 < val_2 ?  val_ :  val_2;
```

Prove that the greater and lesser values of two values `val_1` and `val_2` may be assigned to variables named greater and lesser respectively by one statement as follows:

```
(greater = (lesser = val_1 < val_2?  val_1 :  val_2) ==
val_1)?  val_2 :  val_1;
```

which of the pairs of parentheses above are necessary?

Our next example applies the ternary operator to encode the Russian Peasant Multiplication Algorithm. Apparently peasants in Russia use the following algorithm when they multiply two integers, say `val_1` and `val_2`. (For the sake of computational efficiency, it is necessary to determine the lesser of the multiplier and the multiplicand which from the product.)

Let L be the lesser and G the greater of `val_1` and `val_2`, and let P be the variable to store their product, initialised to 0.

```
while (L is not equal to zero)
    {
     if L is odd, P = P + G;
     Halve L, ignoring any remainder;
     double G;
    }
```

For example, to multiply 19 with 31, L = 19, G = 31, P = 0. Then:

P = 31
L = 9
G = 62
P = 93
L = 4
G = 124
P = 93
L = 2
G = 248
P = 93
L = 1
G = 496
P = 589
L = 0

and the product is 589.

The program in Listing 12 below illustrates the use of the ternary operator to encode the Russian Peasant Algorithm:

```
/* Program 5.14; file name:unit5-prog14.c */
#include <stdio.h>
int main()
{   int val_1, val_2, lesser, greater, result = 0;
    printf("Russian Peasant Multiplication Algorithm\n");
    printf("\nEnter multiplier: ");
    scanf("%d", &val_1);
    printf("\nEnter multiplicand: ");
    scanf("%d", &val_2);
```

```
    greater = (lesser = val_1 < val_2 ? val_1 : val_2)
        == val_1 ? val_2 : val_1;
    while (lesser)
        {
         result += lesser % 2 ? greater : 0;
         lesser /= 2;
         greater *= 2;
        }
    printf("%d\n", result);
    return (0);
}
```

Listing 12: Russian Peasant Algorithm.

E9) Write a C program which uses the ternary operator to print −1, 0 or 1 if the value input to it is negative, zero or positive.

E10) Execute the program listed below, and explain its output.

```
/* Program 5.15; file name: unit5-prog15.c */
#define TRUE 1
#define FALSE 0
#define T "true"
#define F "false"
#include <stdio.h>
int main()
{   int i = FALSE, j = FALSE;
    while (i <= TRUE)
        {
         while (j <= TRUE)
             {
              printf("%s && %s equals %s\n", i ? T : F,
j ? T : F, i && j ? T : F);
              j++;
             }
        i++;
        j = FALSE;
        }
    return (0);
}
```

(Note that the Program 5.15 contains two **while()** loops, one nested fully inside the other. The outer loop is executed for each value of `i`, that i, twice: once when `i` is **FALSE**, and once when it is **TRUE**; the inner loop is executed for each value of `j`. When `i` is **FALSE**, `j` is **FALSE** and **TRUE**; that makes for two executions of the inner loop. When `i` is **TRUE**, `j` again ranges from **FALSE** to **TRUE**, which makes for two further iterations of the inner loop.)

E11) Write a C language program to print a truth table for the Boolean expression `i && (j || k)`, where `i, j` and `k` range from **FALSE** to **TRUE**.

We end this section here. In the next section we will discuss the **switch - case - default** statement.

5.6 THE switch - case - default STATEMENT

The **switch - case - default** statement is useful where control flow within a program must be directed to one of several possible execution paths. We've already seen that a chain of **if() - else** may be used in this sort of situation. But **if() - else** become cumbersome when the number of alternative exceeds three or four; then the **switch -**

case - default statement is appropriate. For an illustrative program, let's write one to answer questions such as: What day was it on 15 August 1947? What day will it be on NEW Year's Day in AD 4000? There's a famous algorithm, called Zeller's congruence, that one can use to find the day for any valid date between 1581 Ad and 4902 AD:

Step 1: January and February are considered the eleventh and twelfth months of the preceding year; March, April, May, ... , December are considered the first, second, third, ..., tenth months of the current year. For example, to find the day which fell on 23 January 1907, the Month number must be set at 11, the Year at 1906.

Step 2: Then, given the values of Day, Month and Year, from Step 1 the expression:

```
Zeller = ((int) ((13 * Month - 1) / 5) + Day + Year % 100 +
(int) ((Year % 100) / 4) - 2 * (int) (year / 100) +
(int) (Year / 400) + 91) % 7
```

will have one of the values 0, 1, 2, 3, ..., 6.
(Why ?)

Step 3: 0 corresponds to a Sunday, 1 to a Monday, etc. and 6 corresponds to a Saturday.

Let's first consider Step 3 of our program. The **if() - else** way to handle it is unwieldy:

```
if (Zeller == 0)
    printf("That was a Sunday.\n");
else
    if(Zeller == 1)
        printf("That was a Monday\n");
    else
        if()...
```

There is the very real danger of further conditions and object statements overflowing the right margin of the page!

In the **switch - case - default** statement, a discrete variable or expression is enclosed in parentheses following the keyword **switch**, and the cases, each governed by one or more distinct values of the switch variable, are listed in a set of curly braces as below:

```
switch (discrete_variable or expression)
    {
     case val_1  :  statements;
                         break;
     case val_2  :  statements;
                         break;
     case val_3  :  statements;
                         break;
              ...
     case val_n  :  statements;
                         break;
     default     :  statements;
                         break;
     }
```

To use the switch statement to encode Zeller's algorithm, for example, one would write:

```
switch (Zeller)
    {
     case 0: printf("\%d-\%d-\%d was a Sunday.\n",
Day, Month, Year);
                                                  break;
     case 1: printf("\%d-\%d-\%d was a Monday.\n",
Day, Month, Year);
                                                  break;
```

```
     case 2: printf("\%d-\%d-\%d was a Tuesday.\n",
Day, Month, Year);
                                        break;
     case 3: printf("\%d-\%d-\%d was a Wednesday.\n",
Day, Month, Year);
                                        break;
     case 4: printf("\%d-\%d-\%d was a Thursday.\n",
Day, Month, Year);
                                        break;
     case 5: printf("\%d-\%d-\%d was a Friday.\n",
Day, Month, Year);
                                        break;
     case 6: printf("\%d-\%d-\%d was a Saturday.\n",
Day, Month, Year);
                                        break;
    }
```

Then, when the **switch** is entered, depending upon the value of `discrete_variable`, the corresponding set of statements (following the keyword case) is executed. But if the switch variable has a value different from those listed with any of the cases, the statements corresponding to the keyword **default** are executed. The **default** case is optional. If it does not occur in the **switch** statement, and if the value obtained by the **switch** variable does not correspond to any listed in the **cases**, the statement is skipped in its entirely. We don't have a default statement in the example above: there's no way that an **int** value can leave a remainder outside the set [0 - 6] on division by 7!

The **break** statement, the last statement of every case including the default is **necessary**! If it is not there, control would fall through to execute the next case. This property of the case may useful when multiple values of the `discrete_variable` say `val_1`, `val_2` or `val_3` must trigger the same set of statements:

```
case val_1 :
case val_2 :
case val_3 :  statements;
                  break;
```

If `discrete_variable` happens to get any of the values `val_1, val_2` or `val_3`, control will reach the corresponding case and fall through to the set of statements corresponding to `val_3`. Though more than one **case** value may set off the same set of statements, **case** values must all be different; and they may be in any order.

Providing a **break** after the statements listed against **default** ensures that if you add another case after the program is up and running, and `discrete_variable` happens to get a value corresponding to **default**, control will not then flow forwards to execute the case you added last! The **break** statement must particularly be kept in mind by Pascal programmers writing C programs, because the equivalent Pascal statement does not need such a mechanism to escape from the cases.

Probably the most surprising line of Listing 13 on the following page is the **scanf()** statement written as a component of the Boolean controlling the **while()** loop:

```
while (scanf("%d-%d-%d", &Day, &Month, &Year) ! = 3)
```

Here we have made used of the **scanf()** property that it returns the number of values read and assigned, which in this case must be three: for Day, Month and Year. The **while()** loop is executed until the expected number and types of values for these variables have been entered. Note also that the **scanf()** contains non-format character—the hyphens—in its control string: it expects the values of Day, Month and Year to be separated by single hyphens, *and no other characters*. These, and other properties of **scanf()** are discussed in Unit 7.

An important point to note in the program below are the tests to determine whether a date entered in valid. For Zeller's congruence to work correctly, no year value can lie outside the range 1582–4902; moreover, no month number can be less than one, or greater that twelve; no month can have more days than 31, and no date in a month can have a value less than 1; February can have 29 days only in a leap year; and February, April, June, September and November have less than 31 days apiece.

Note also that we have chosen to introduce two additional **int** variables `ZMonth` and `ZYear` which are assigned values by the **if()** statements below:

```
if (Month < 3)
    ZMonth = Month + 10;
else
    ZMonth = Month - 2;
if (ZMonth < 10)
    ZMonth = Year - 1;
else
    ZYear = year;
```

Observe that these statements are in accord with Step 1 of Zeller's Algorithm.

It is always good policy to retain the values of input variables; since the formula calls for changed values of `Month` and `Year`, we store the new values in `ZMonth` and `ZYear`, leaving `Month` and `Year` untouched. That way, they'll be available when we need their values later, in the **printf()** statements with the cases.

Finally, the **exit()** function is useful when immediate program termination is required. The call

```
exit(n)
```

closes all files that may be open, flushes memory buffers, and returns the value n, the exit status, to be function that called it; in a Unix like programming environment, n is usually zero for error free termination, non-zero otherwise; the value of n may be used to diagnose the error.

```
/* Program 5.16; file name: unit5-prog16.c*/
#include <stdio.h>
#include <stdlib.h>
#define LEAP_YEAR (Year % 4 == 0 && Year % 100 != 0)\
|| Year % 400 == 0
int main()
{   int Day, Month, Year, Zeller, ZMonth, ZYear;
    printf("\nThis program finds the day \
corresponding to a given date.\n");
    printf("\nEnter date, month, year ... \
format is dd-mm-yyyy.\n");
    printf("\nEnter a 1 or 2-digit number \
for day, followed by a\n");
    printf("\ndash, followed by a 1 or 2-digit \
number for month,\n");
    printf("\nfollowed by a dash, followed by a \
2 or 4-digit number\n");
    printf("\nfor the year.  Valid year range \
is 1582-4902, inclusive.\n");
    printf("\n(A 2-digit number for the year will \
imply 20th century\n");
    printf("\nyears.)\n\n\n\n Enter dd-mm-yyyy: ");
    while (scanf("%d-%d-%d", &Day, &Month, &Year) != 3)
        {
         printf("\nInvalid number of arguments \
or hypens mismatched\n");
```

```
        }
    if (Year < 0)
        {
         printf("\nInvalid year value...Program aborted..");
         exit (1);
        }
    if (Year < 100)
        Year += 1900;
    if (Year < 1582 || Year > 4902)
        {
         printf("\nInvalid year value...Program aborted..");
         exit(1);
        }
    if (!(LEAP_YEAR) && (Month == 2) && (Day > 28))
        {
         printf("\nInvalid date...Program aborted..");
         exit(1);
        }
    if ((LEAP_YEAR) && (Month == 2) && (Day > 29))
        {
         printf("\nInvalid date...Program aborted..");
         exit(1);
        }
    if (Month < 1 || Month > 12)
        {
         printf("\nInvalid month...Program aborted..");
         exit(1);
        }
    if (Day < 1 || Day > 31)
        {
         printf("\nInvalid date...Program aborted..");
         exit(1);
        }
    if ((Day > 30) && (Month == 4 || Month == 6 ||
Month == 9 || Month == 11))
        {
         printf("\nInvalid date...Program aborted..");
         exit(1);
        }
    if (Month < 3)
        ZMonth = Month + 10;
    else
        ZMonth = Month - 2;
        if (ZMonth > 10)
            ZYear = Year - 1;
        else
            ZYear = Year;

    Zeller = ((int) ((13 * ZMonth - 1) / 5) + Day + ZYear % 100 +
    (int) ((ZYear % 100) / 4) - 2 * (int) (ZYear / 100) +
    (int) (ZYear / 400) + 91) % 7;
    printf("\n\n\n                                                ");
    switch (Zeller)
        {
         case 0:
         printf("%d-%d-%d was a Sunday.\n", Day, Month, Year);
         break;
         case 1:
         printf("%d-%d-%d was a Monday.\n", Day, Month, Year);
         break;
         case 2:
         printf("%d-%d-%d was a Tuesday.\n", Day, Month, Year);
```

```
            break;
            case 3:
            printf("%d-%d-%d was a Wednesday.\n", Day, Month, Year);
            break;
            case 4:
            printf("%d-%d-%d was a Thursday.\n", Day, Month, Year);
            break;
            case 5:
            printf("%d-%d-%d was a Friday.\n", Day, Month, Year);
            break;
            case 6:
            printf("%d-%d-%d was a Saturday.\n", Day, Month, Year);
            break;
           }
    return (0);
}
```

Listing 13: Zeller's formula.

A long replacement string of a `#define` may be continued into the next line by placing a backslash at the end of the part of the string in the current line. (Reread the third and fourth lines of Listing 13.)

> **Note**
>
> Recall that we can use this device with quoted strings: if you have to deal with a long string, longer than you can conveniently type in a single line, you may split it over several lines by placing a backslash as the last character of the part in the current line. For example:
> "A long string, continued over two lines using the backslash \
> character."
> ANSI C treats string constants separated by white space characters as an unbroken string.

E12) In Listing 13 why is it preferable to write

```
if ((Day > 30) && (Month == 4 || Month == 6 || Month == 9
 || Month == 11))
```

etc. instead of

```
if ((Month == 4 || Month == 6 || Month == 9 || Month ==
11) && (Day > 30))
```

etc?

E13) Is the cast operator (**int**) required in the expression for Zeller in Listing 13? Rewrite the expression without the case operator, and without redundant parentheses.

E14) Since 91 is exactly divisible by 7, is its presence required in the expression for Zeller in Listing 13? Explain why or why not.

E15) In Listing 13, is `!(LEAP_YEAR)` different from `!LEAP_YEAR`?

E16) The **switch - case - default** statement is not always a better choice than the **if() - else**, even when there may be several cases to include in a program. Rewrite the following program, which scans a **char** input and determines its hexadecimal value (if it has one) by using a **switch** instead of the **if() - else**s.

```
/* Program 5.17 */
#include <stdio.h>
int main()
```

```
{
        char digit;
    printf("Enter hex digit: ");
    scanf("%c", &digit);
    if (digit >= 'a' && digit <= 'f')
            printf("Decimal value of hex digit \
is %d.\n", digit = digit - 'a' + 10);
        else
        if (digit >= 'A' && digit <= 'F')
            printf("Decimal value of hex digit \
is %d.\n", digit =
            digit - 'A' + 10);
        else
            if (digit >= '0' && digit <= '9')
                printf("Decimal value of hex digit \
is %d.\n", digit - '0');
            else
                printf("You typed a non-hex digit.\n");
        return (0);
}
```

E17) Compile and execute the program below and state its output:

```
/* Program 5.18; file name: unit5-prog18.c */
#include <stdio.h>
int main()
{
    printf("%s", "How" "many" "strings" "do" "we" "have?");
    return (0);
}
```

In Program 5.17 note that the alphabetical hex digits may be entered both in lowercase or in uppercase characters. But this makes for a long **if( ) - else**:

```
if(digit = 'a' && digit 'f')
        etc
else
    if (digit = 'A' && digit 'F')
        etc..
```

The `toupper()` function may be used with advantage in such situation; it returns the uppercase equivalent of its character argument (if it was a lowercase character) or its argument itself if it wasn't. The values of its argument remains unaltered. Thus `toupper('x')` returns `'X', toupper` ('?') returns'?'. To use the `toupper`() function, **#include** the <ctype.h> just as you do the file <stdio.h>. See Listing 14 below.

```
/* Program 5.19; file name: unit5-prog19.c */
#include <stdio.h>
#include <ctype.h>
int main ()
{
    char digit;
    printf("Enter hex digit:");
    scanf("%c", &digit);
    if(toupper(digit) >= 'A' && toupper (digit) <= 'F')
        printf("Decimal value of hex digit is %d.\n",
        digit >= 'a' ?digit - 'a' + 10: digit - 'A' + 10);
    else
        if (digit >= '0' && digit <= '9')
            printf("Decimal value of hex digit \
is %d.\n", digit - '0');
        else
```

```
                printf("You typed a non-hex digit.\n");
    return (0);
}
```

Listing 14: Using function toupper().

Besides the functions `toupper`() and its analogue called `tolower(), ctype,h` provides many other functions for testing characters which you may often find useful. These functions return a non-zero (**true**) value if the argument satisfies the stated condition:

isalnum	(c)	c is an alphabetic or numeric character
isalpha	(c)	c is an alphabetic character
iscntrl	(c)	c is a control character
isdigit	(c)	c is a decimal digit
isgraph	(c)	c is a graphics character
islower	(c)	c is a lowercase character
isprint	(c)	c is a printable character
ispunet	(c)	c is a punctuation character
isspace	(c)	c is a space, horizontal or vertical tab, formfeed, newline or carriage return character
isupper	(c)	c is an uppercase character
isxdigit	(c)	c is hexadecimal digit

Note that the <ctype.h> function `isxdigit (c)` makes Program 5.17 and 5.19 somewhat superfluous!

In the program in Listing 15 below we use the **switch** statement to count spaces (blanks or horizontal tabs), punctuation marks, vowels (both upper and lowercase), lines and the total number of keystrokes received. The program processes input until it is sent a character that signifies and of input. This character is customarily called the "end of file" character, or EOF, but it's not really a character: for if EOF is to signify end of input to a program, its value must be different from that of any **char**. That is why the variable c is Program 5.19 is declared an **int; int**s can include all chars in their range, as well as the EOF. A MACRO definition in <stdio.h> **#define**s a value for it. So **#includ**ing <stdio.h> makes EOF automatically avail able to your program.

In the **while()** statement:

```
while ((c = getchar ()) ! = EOF)
```

note that

```
c = getchar ()
```

is performed before c is compared against EOF. c first gets a value; that value is then compared against EOF. As long as c is different from EOF, the Boolean controlling the **while()** remains **true**, and the loop is executed. When EOF is encountered, the loop is exited, and the program terminates. In the DOS environment on a PC, EOF is sent by pressing the CTRL and Z keys together. In the bash shell in linux EOF is sent by pressing the CTRL and D keys together.

```
/* Program 5.20; file name:unit5-prog20.c */
#include <stdio.h>
int main ()
{   int c;
    long keystrokes = 0, spaces = 0, punct_marks = 0,
    lines = 0, vowels = 0;
    printf("Enter text, line by line, and I\'ll give you some\n \
statistics about it...terminate your input by entering\n\
CTRL-Z as the first character of a newline...that\'s the\n\
DOS EOF character (may be different in your computing \
enviornment);\n\n");
```

```
    while ((c = getchar()) !=  EOF)
        {
         switch(c)
             {
              case '\n':  ++ lines;
                          keystrokes ++;
                          break;
              case '\t':
              case ' ': spaces ++;
                        keystrokes ++;
                        break;
              case ',':
              case '.':
              case ':':
              case ';':
              case '!':
              case '?': punct_marks ++;
                        keystrokes ++;
                        break;
              case 'a':
              case 'A':
              case 'e':
              case 'E':
              case 'i':
              case 'I':
              case 'o':
              case 'O':
              case 'u':
              case 'U':
              case 'y':
              case 'Y': vowels ++;
                        keystrokes ++;
                        break;
              default:
                        keystrokes ++;
                        break;
             }
        }
    printf("Input statistics:\
\n lines = %ld,\n keystrokes = %ld,\n vowels = %ld,\
\n punctuation marks = %ld, spaces = %ld\n", lines,
keystrokes, vowels, punct_marks, spaces);
    return (0);
}
```

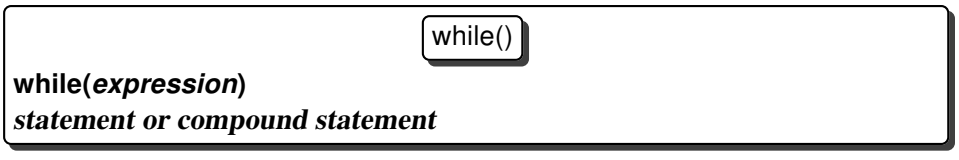
Listing 15: Program to count the number of charaters.

The **switch** statement is useful only when the cases are controlled by integer values; in a program where the conditions to control branching are set in terms of floating point values, the **if() - else** statement must be used.

5.7 SUMMARY

In this unit we have

1. discussed the **while()** and **do-while()** loops.



do-while()

do
statement or compound statement
while()

In the **while()** loop the expression is evaluated **before** entering the loop. In the **do-while()** loop, the expression is evaluated at the end of the loop. So, the **do-while()** loop is used if the loop has to be executed at least once.

2. discussed the ternary operator **if-then-else**

if-then-else

```
x = condition ?  first_val :  second_val;
```

In this case the operator tests the condition and return one of the two values, the first if the condition is true and the second if it is not.

3. discussed the comma operator than can be used to control the order of evaluation of expressions.

5.8 SOLUTIONS/ANSWERS

E1)
```
/*File name: unit5-ans-ex1.c*/
#include <stdio.h>
int main()
{
    int a, p, b, i = 1;
    printf("Enter p,a\n");
    scanf("%d,%d",&p,&a);
    b = a;
    while (a != 1)
        {
         a *= b;
         a %= p;
         i++;
        }
    printf("The order is %d.",i);
    return (0);
}
```

E2)
```
/* File name: unit5-ans-ex2.c */
#include <stdio.h>
int main()
{   int fib1, fib2, pairs, loop_index = 3, month;
        printf("Rabbit pairs produced in which month?...: ");
    scanf("%d", &month);
    if (month == 1 || month == 2)
        pairs = 1;
    else
        {
         fib1 = fib2 = 1;
         do
             {
              pairs = fib1 + fib2;
              fib1 = fib2;
              fib2 = pairs;
```

```
            }
               while( ++loop_index <= month);

        }/*End else*/
    printf("The number of pairs produced in \
month %d is %d.\n", month,pairs);
    return (0);
}
```

E3)
```
/*File name:unit5-ans-ex3.c*/
#include <stdio.h>
#define LIMIT 25000
int main ()
{
    int sum = 1, i=1;
    while (sum <= LIMIT)
        {
         i++;
         sum += i*i;
        }
        printf("The sum exceeds %d when \
i = %d", LIMIT, i);
        return (0);
}
```

E4) Observe that the number itself is divisible by 11 and must be a square. The small number divisible by 11 whose square is also a 4 digit number is 33.

E5)
```
/* Program 5.12; file name:unit5-ans-ex4.c */
#include <stdio.h>
int main ()
        {
     int a = 1, b, sqroot;
     while (a <= 9)
         {
          b = 1;
          while (b <= 9)
              {
               sqroot = 32;
               while (sqroot ++ < 99)
                   {
                    if (sqroot * sqroot <
                        1100 * a + 11 * b)
                        continue;
                    else
                        if(sqroot * sqroot ==
                            1100 * a + 11 * b)
                            printf("Car\'s number is : \
%d\n", 1100 * a + 11 * b);
                        else
                            break;
                   }
                b ++;
              }
           a ++;
         }
    return (0);
}
```

E6) Since the value of the expression is 2, it will print the first statement.

E7) As mentioned, `val_1 < val_2, lesser` will get the value `val_1`, otherwise, it will get the value `val_2`. If the value of lesser is `val_1 lesser = val_1 < val_2? val_1 : val_2) == val_1` will have the value 1, so `greater` will get the value `val_2`. Otherwise, the value will be zero, i.e. `val_1` is the bigger number and `greater` will get the value `val_1`. The outer parentheses are not necessary.

E8)
```
#include <stdio.h>
int main ()
{   int a, b;
    printf("Enter a number...\n");
    scanf("%d",&a);
    b = (a < 0?-1:1)*(a == 0?0:1);
    printf("%d",b);
    return (0);
}
```

E9) This program outputs the truth table of the Boolean expression `i&&j` where `i, j` range from **FALSE** to **TRUE**.

E10)
```
/* Program 5.15; file name: unit5-prog15.c */
#define TRUE 1
#define FALSE 0
#define T "true"
#define F "false"
#include <stdio.h>
int main()
{   int i = FALSE, j = FALSE, k = FALSE;
    while (i <= TRUE)
        {
         while (j <= TRUE)
             {
              while(k <= TRUE)
                  {
                   printf("%s && (%s || %s) equals %s\n",
i ? T : F, j ? T : F, k? T : F,i && (j||k) ? T : F);
                   k++;
                  }
               k = FALSE;
               j++;
             }
        i++;
        j = FALSE;
        }
    return (0);
}
```

E11) If Day is less than 30, in the first case, the program will not check the month. In the second case, it will check the month and the the Day also.

E12)
```
Zeller = ((13 * ZMonth - 1) / 5 + Day + ZYear % 100 +
ZYear % 100) / 4 - 2 *  (ZYear / 100) +
(ZYear / 400) + 91) % 7
```

E13) 91 is there to make sure that the answer is never negative. In the number is negative % operator gives negative answer. The worst case is Month $= 2$, Year $= 4900$ and Day $= 1$.The value of the expression will be -90 without the added multiple of 7. The smallest multiple of 7 bigger than -90 is 91 and we have added this number.

E14) Yes. In the first case, `!(LEAP_YEAR)`, `!` is applied to the entire expression. In the second case, it is applied just to the first term,
`(Year % 4 == 0 && Year % 100 != 0)` and so the expression becomes
`!(Year % 4 == 0 && Year % 100 != 0)|| Year % 400 == 0` which means the remainder of Year on division by 4 is 0 or the remainder of Year on division by 400 is 0!

E15) Try it! It will have 32 cases if you do not use **`if ()`** or any other decision structure.

E16) **`Howmanystringsdowehave?`**