1 Getting Started

- What is C
- Getting Started with C
  - The C Character Set
  - Constants, Variables and Keywords
  - Types of C Constants
  - Rules for Constructing Integer Constants
  - Rules for Constructing Real Constants
  - Rules for Constructing Character Constants
  - Types of C Variables
  - Rules for Constructing Variable Names
  - C Keywords
- The First C Program
- Compilation and Execution
- Receiving Input
- C Instructions
  - Type Declaration Instruction
  - Arithmetic Instruction
  - Integer and Float Conversions
  - Hierarchy of Operations
  - Associativity Of Operators
- Control Instruction in C
- Summary
- Exercise

Before we can begin to write serious programs in C, it would be interesting to find out what really is C, how it came into existence and how does it compare with other computer languages. In this chapter we would briefly outline these issues.

Four important aspects of any language are the way it stores data, the way it operates upon this data, how it accomplishes input and output and how it lets you control the sequence of execution of instructions in a program. We would discuss the first three of these building blocks in this chapter.

What is C

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc. No one pushed C. It wasn't made the 'official' Bell Labs language. Thus, without any advertisement C's reputation spread and its pool of users grew. Ritchie seems to have been rather surprised that so many programmers preferred C to older languages like FORTRAN or PL/I, or the newer ones like Pascal and APL. But, that's what happened.

Possibly why C seems so popular is because it is reliable, simple and easy to use. Moreover, in an industry where newer languages, tools and technologies emerge and vanish day in and day out, a language that has survived for more than 3 decades has to be really good.

An opinion that is often heard today is – "C has been already superceded by languages like C++, C# and Java, so why bother to

learn C today". I seriously beg to differ with this opinion. There are several reasons for this:

(a) I believe that nobody can learn C++ or Java directly. This is because while learning these languages you have things like classes, objects, inheritance, polymorphism, templates, exception handling, references, etc. do deal with apart from knowing the actual language elements. Learning these complicated concepts when you are not even comfortable with the basic language elements is like putting the cart before the horse. Hence one should first learn all the language elements very thoroughly using C language before migrating to C++, C# or Java. Though this two step learning process may take more time, but at the end of it you will definitely find it worth the trouble.

(b) C++, C# or Java make use of a principle called Object Oriented Programming (OOP) to organize the program. This organizing principle has lots of advantages to offer. But even while using this organizing principle you would still need a good hold over the language elements of C and the basic programming skills.

(c) Though many C++ and Java based programming tools and frameworks have evolved over the years the importance of C is still unchallenged because knowingly or unknowingly while using these frameworks and tools you would be still required to use the core C language elements—another good reason why one should learn C before C++, C# or Java.

(d) Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C.

(e) Mobile devices like cellular phones and palmtops are becoming increasingly popular. Also, common consumer devices like microwave oven, washing machines and digital cameras are getting smarter by the day. This smartness comes from a microprocessor, an operating system and a program embedded in this devices. These programs not only have to run fast but also have to work in limited amount of memory. No wonder that such programs are written in C. With these constraints on time and space, C is the language of choice while building such operating systems and programs.

(f) You must have seen several professional 3D computer games where the user navigates some object, like say a spaceship and fires bullets at the invaders. The essence of all such games is speed. Needless to say, such games won't become popular if they takes a long time to move the spaceship or to fire a bullet. To match the expectations of the player the game has to react fast to the user inputs. This is where C language scores over other languages. Many popular gaming frameworks have been built using C language.

(g) At times one is required to very closely interact with the hardware devices. Since C provides several language elements that make this interaction feasible without compromising the performance it is the preferred choice of the programmer.

I hope that these are very convincing reasons why one should adopt C as the first and the very important step in your quest for learning programming languages.

Getting Started with C

Communicating with a computer involves speaking the language the computer understands, which immediately rules out English as the language of communication with computer. However, there is

a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form a program. This is illustrated in the Figure 1.1.

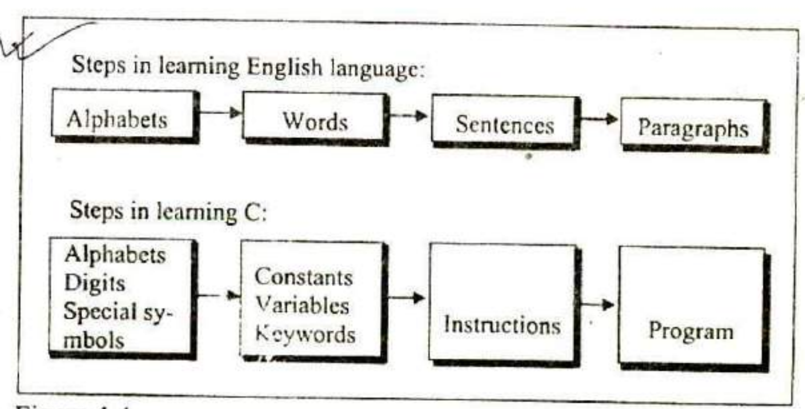


Figure 1.1

The C Character Set

A character denotes any alphabet, digit or special symbol used to represent information. Figure 1.2 shows the valid alphabets, numbers and special symbols allowed in C.

<table>
<tr><td>Alphabets</td><td>A, B, ..., Y, Z
a, b, ..., y, z</td></tr>
<tr><td>Digits</td><td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9</td></tr>
<tr><td>Special symbols</td><td>~ ' ! @ # % ^ & * ( ) _ - + = | \ { }
[ ] : ; " ' <> , . ? /</td></tr>
</table>

Figure 1.2

Constants, Variables and Keywords

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords. Let us see what are 'constants' and 'variables' in C. A constant is an entity that doesn't change whereas a variable is an entity that may change.

In any program we typically do lots of calculations. The results of these calculations are stored in computers memory. Like human memory the computer memory also consists of millions of cells. The calculated values are stored in these memory cells. To make the retrieval and usage of these values easy these memory cells (also called memory locations) are given names. Since the value stored in each location may change the names given to these locations are called variable names. Consider the following example.

Here 3 is stored in a memory location and a name **x** is given to it. Then we are assigning a new value 5 to the same memory location **x**. This would overwrite the earlier value 3, since a memory location can hold only one value at a time. This is shown in Figure 1.3.

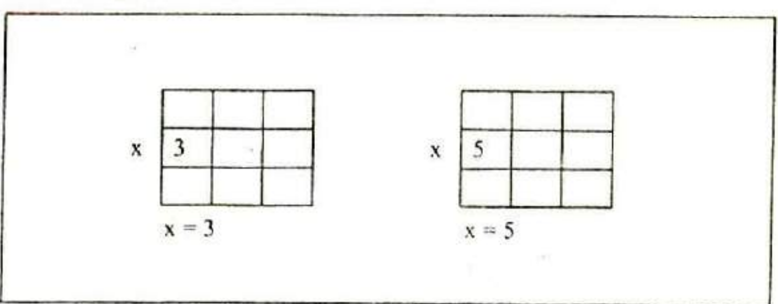


Figure 1.3

Since the location whose name is **x** can hold different values at different times **x** is known as a variable. As against this, 3 or 5 do not change, hence are known as constants.

Types of C Constants

C constants can be divided into two major categories:

(a) Primary Constants
(b) Secondary Constants

These constants are further categorized as shown in Figure 1.4.

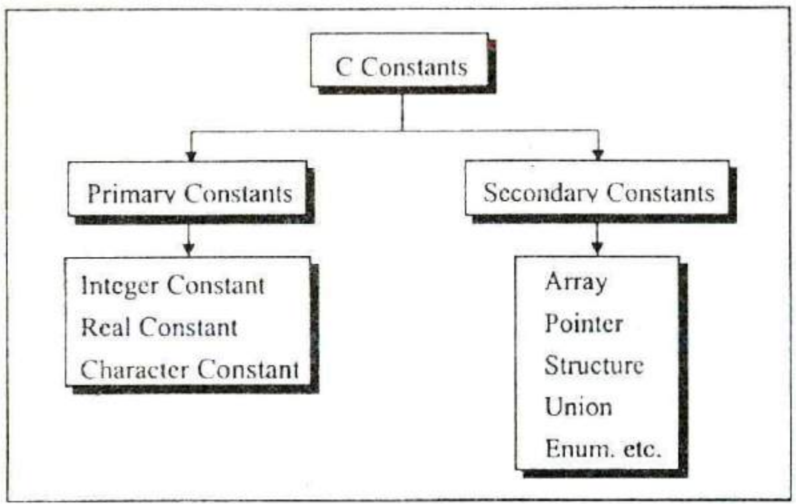


Figure 1.4

At this stage we would restrict our discussion to only Primary Constants, namely, Integer, Real and Character constants. Let us see the details of each of these constants. For constructing these different types of constants certain rules have been laid down. These rules are as under:

Rules for Constructing Integer Constants

(a) An integer constant must have at least one digit.
(b) It must not have a decimal point.
(c) It can be either positive or negative.
(d) If no sign precedes an integer constant it is assumed to be positive.
(e) No commas or blanks are allowed within an integer constant.
(f) The allowable range for integer constants is -32768 to 32767.

Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the

range is –32768 to 32767. For a 32-bit compiler the range would be even greater. Question like what exactly do you mean by a 16-bit or a 32-bit compiler, what range of an Integer constant has to do with the type of compiler and such questions are discussed in detail in Chapter 16. Till that time it would be assumed that we are working with a 16-bit compiler.

```
Ex.: 426
     +782
     -8000
     -7605
```

Rules for Constructing Real Constants

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.

Following rules must be observed while constructing real constants expressed in fractional form:

(a) A real constant must have at least one digit.
(b) It must have a decimal point.
(c) It could be either positive or negative.
(d) Default sign is positive.
(e) No commas or blanks are allowed within a real constant.

```
Ex.: +325.34
     426.0
     -32.76
     -48.5792
```

The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large. It however doesn't restrict us in any way from using exponential form of representation for other real constants.

In exponential form of representation, the real constant is represented in two parts. The part appearing before 'e' is called mantissa, whereas the part following 'e' is called exponent.

Following rules must be observed while constructing real constants expressed in exponential form:

(a) The mantissa part and the exponential part should be separated by a letter e.
(b) The mantissa part may have a positive or negative sign.
(c) Default sign of mantissa part is positive.
(d) The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.
(e) Range of real constants expressed in exponential form is -3.4e38 to 3.4e38.

```
Ex.: +3.2e-5
     4.1e8
     -0.2e+3
     -3.2e-5
```

Rules for Constructing Character Constants

(a) A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to the left. For example, 'A' is a valid character constant whereas 'A' is not.
(b) The maximum length of a character constant can be 1 character.

```
Ex.: 'A'
     'I'
     '5'
     '='
```

Types of C Variables

As we saw earlier, an entity that may vary during program execution is called a variable. Variable names are names given to locations in memory. These locations can contain integer, real or character constants. In any language, the types of variables that it can support depend on the types of constants that it can handle. This is because a particular type of variable can hold only the same type of constant. For example, an integer variable can hold only an integer constant, a real variable can hold only a real constant and a character variable can hold only a character constant.

The rules for constructing different types of constants are different. However, for constructing variable names of all types the same set of rules apply. These rules are given below.

Rules for Constructing Variable Names

(a) A variable name is any combination of 1 to 31 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 247 characters. Still, it would be safer to stick to the rule of 31 characters. Do not create unnecessarily long variable names as it adds to your typing effort.

(b) The first character in the variable name must be an alphabet or underscore.

(c) No commas or blanks are allowed within a variable name.

(d) No special symbol other than an underscore (as in **gross_sal**) can be used in a variable name.

```
Ex.: si_int
     m_hra
     pop_e_89
```

These rules remain same for all the types of primary and secondary variables. Naturally, the question follows... how is C able to differentiate between these variables? This is a rather simple

matter. C compiler is able to distinguish between the variable names by making it compulsory for you to declare the type of any variable name that you wish to use in a program. This type declaration is done at the beginning of the program. Following are the examples of type declaration statements:

```
Ex.: int  si, m_hra ;
     float  bassal ;
     char  code ;
```

Since, the maximum allowable length of a variable name is 31 characters, an enormous number of variable names can be constructed using the above-mentioned rules. It is a good practice to exploit this enormous choice in naming variables by using meaningful variable names.

Thus, if we want to calculate simple interest, it is always advisable to construct meaningful variable names like **prin**, **roi**, **noy** to represent Principle, Rate of interest and Number of years rather than using the variables **a**, **b**, **c**.

C Keywords

Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). The keywords **cannot** be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. Some C compilers allow you to construct variable names that exactly resemble the keywords. However, it would be safer not to mix up the variable names and the keywords. The keywords are also called 'Reserved words'.

There are only 32 keywords available in C. Figure 1.5 gives a list of these keywords for your ready reference. A detailed discussion of each of these keywords would be taken up in later chapters wherever their use is relevant.

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

Figure 1.5

Note that compiler vendors (like Microsoft, Borland, etc.) provide their own keywords apart from the ones mentioned above. These include extended keywords like **near**, **far**, **asm**, etc. Though it has been suggested by the ANSI committee that every such compiler specific keyword should be preceded by two underscores (as in **__asm**), not every vendor follows this rule.

The First C Program

Armed with the knowledge about the types of variables, constants & keywords the next logical step is to combine them to form instructions. However, instead of this, we would write our first C program now. Once we have done that we would see in detail the instructions that it made use of.

Before we begin with our first C program do remember the following rules that are applicable to all C programs:

(a) Each instruction in a C program is written as a separate statement. Therefore a complete C program would comprise of a series of statements.

(b) The statements in a program must appear in the same order in which we wish them to be executed; unless of course the logic of the problem demands a deliberate 'jump' or transfer of control to a statement, which is out of sequence.

(c) Blank spaces may be inserted between two words to improve the readability of the statement. However, no blank spaces are allowed within a variable, constant or keyword.

(d) All statements are entered in small case letters.

(e) C has no specific rules for the position at which a statement is to be written. That's why it is often called a free-form language.

(f) Every C statement must end with a ;. Thus ; acts as a statement terminator.

Let us now write down our first C program. It would simply calculate simple interest for a set of values representing principle, number of years and rate of interest.

```
/* Calculation of simple interest */
/* Author gekay  Date: 25/05/2004 */
main( )
{
    int  p, n ;
    float  r, si ;

    p = 1000 ;
    n = 3 ;
    r = 8.5 ;

    /* formula for simple interest */
    si = p * n * r / 100 ;

    printf ( "%f" , si ) ;
```

```
}
```

Now a few useful tips about the program...

- Comment about the program should be enclosed within /* */. For example, the first two statements in our program are comments.

- Though comments are not necessary, it is a good practice to begin a program with a comment indicating the purpose of the program, its author and the date on which the program was written.

- Any number of comments can be written at any place in the program. For example, a comment can be written before the statement, after the statement or within the statement as shown below:

```
/* formula */ si = p * n * r / 100 ;
si = p * n * r / 100 ; /* formula */
si = p * n * r / /* formula */ 100 ;
```

- Sometimes it is not so obvious as to what a particular statement in a program accomplishes. At such times it is worthwhile mentioning the purpose of the statement (or a set of statements) using a comment. For example:

```
/* formula for simple interest */
si = p * n * r / 100 ;
```

- Often programmers seem to ignore writing of comments. But when a team is building big software well commented code is almost essential for other team members to understand it.

- Although a lot of comments are probably not necessary in this program, it is usually the case that programmers tend to use too few comments rather than too many. An adequate number of comments can save hours of misery and suffering when you later try to figure out what the program does.

- The normal language rules do not apply to text written within /* .. */. Thus we can type this text in small case, capital or a combination. This is because the comments are solely given for the understanding of the programmer or the fellow programmers and are completely ignored by the compiler.

- Comments cannot be nested. For example,

```
/* Cal of SI /* Author sam date 01/01/2002 */ */
```

  is invalid.

- A comment can be split over more than one line, as in,

```
/* This is
   a jazzy
   comment */
```

  Such a comment is often called a multi-line comment.

- **main( )** is a collective name given to a set of statements. This name has to be **main( )**, it cannot be anything else. All statements that belong to **main( )** are enclosed within a pair of braces { } as shown below.

```
main( )
{
    statement 1 ;
    statement 2 ;
```

```
    statement 3 ;
}
```

- Technically speaking **main( )** is a function. Every function has a pair of parentheses ( ) associated with it. We would discuss functions and their working in great detail in Chapter 5.

- Any variable used in the program must be declared before using it. For example,

```
int  p, n ;
float  r, si ;
```

- Any C statement always ends with a ;

  For example,

```
float  r, si ;
r = 8.5 ;
```

- In the statement,

```
si = p * n * r / 100 ;
```

  * and / are the arithmetic operators. The arithmetic operators available in C are +, -, * and /. C is very rich in operators. There are about 45 operators available in C. Surprisingly there is no operator for exponentiation... a slip, which can be forgiven considering the fact that C has been developed by an individual, not by a committee.

- Once the value of **si** is calculated it needs to be displayed on the screen. Unlike other languages, C does not contain any instruction to display output on the screen. All output to screen is achieved using readymade library functions. One such

function is **printf( )**. We have used it display on the screen the value contained in **si**.

The general form of **printf( )** function is,

```
printf ( "<format string>", <list of variables> ) ;
```

<format string> can contain,

%f for printing real values
%d for printing integer values
%c for printing character values

In addition to format specifiers like **%f**, **%d** and **%c** the format string may also contain any other characters. These characters are printed as they are when the **printf( )** is executed.

Following are some examples of usage of **printf( )** function:

```
printf ( "%f", si ) ;
printf ( "%d %d %f %f", p, n, r, si ) ;
printf ( "Simple interest = Rs. %f", si ) ;
printf ( "Prin = %d \nRate = %f", p, r ) ;
```

The output of the last statement would look like this...

```
Prin = 1000
Rate = 8.5
```

What is '\n' doing in this statement? It is called newline and it takes the cursor to the next line. Therefore, you get the output split over two lines. '\n' is one of the several Escape Sequences available in C. These are discussed in detail in Chapter 11. Right now, all that we can say is '\n' comes in

handy when we want to format the output properly on separate lines.

printf( ) can not only print values of variables, it can also print the result of an expression. An expression is nothing but a valid combination of constants, variables and operators. Thus, 3, 3 + 2, c and a + b * c – d all are valid expressions. The results of these expressions can be printed as shown below:

```
printf ( "%d %d %d %d", 3, 3 + 2, c, a + b * c – d ) ;
```

Note that **3** and **c** also represent valid expressions.

Compilation and Execution

Once you have written the program you need to type it and instruct the machine to execute it. To type your C program you need another program called Editor. Once the program has been typed it needs to be converted to machine language (0s and 1s) before the machine can execute it. To carry out this conversion we need another program called Compiler. Compiler vendors provide an Integrated Development Environment (IDE) which consists of an Editor as well as the Compiler.

There are several such IDEs available in the market targeted towards different operating systems. For example, Turbo C, Turbo C++ and Microsoft C are some of the popular compilers that work under MS-DOS; Visual C++ and Borland C++ are the compilers that work under Windows, whereas gcc compiler works under Linux. Note that Turbo C++, Microsoft C++ and Borland C++ software also contain a C compiler bundled with them. If you are a beginner you would be better off using a simple compiler like Turbo C or Turbo C++. Once you have mastered the language elements you can then switch over to more sophisticated compilers like Visual C++ under Windows or gcc under Linux. Most of the

programs in this book would work with all the compilers. Wherever there is a deviation I would point it out that time.

Assuming that you are using a Turbo C or Turbo C++ compiler here are the steps that you need to follow to compile and execute your first C program...

(a) Start the compiler at **C>** prompt. The compiler (TC.EXE is usually present in **C:\TC\BIN** directory).
(b) Select **New** from the **File** menu.
(c) Type the program.
(d) Save the program using **F2** under a proper name (say Program1.c).
(e) Use **Ctrl + F9** to compile and execute the program.
(f) Use **Alt + F5** to view the output.

Note that on compiling the program its machine language equivalent is stored as an EXE file (Program1.EXE) on the disk. This file is called an executable file. If we copy this file to another machine we can execute it there without being required to recompile it. In fact the other machine need not even have a compiler to be able to execute the file.

A word of caution! If you run this program in Turbo C++ compiler, you may get an error — "The function printf should have a prototype". To get rid of this error, perform the following steps and then recompile the program.

(a) Select 'Options' menu and then select 'Compiler | C++ Options'. In the dialog box that pops up, select 'CPP always' in the 'Use C++ Compiler' options.
(b) Again select 'Options' menu and then select 'Environment | Editor'. Make sure that the default extension is 'C' rather than 'CPP'.

Receiving Input

In the program discussed above we assumed the values of **p**, **n** and **r** to be 1000, 3 and 8.5. Every time we run the program we would get the same value for simple interest. If we want to calculate simple interest for some other set of values then we are required to make the relevant change in the program, and again compile and execute it. Thus the program is not general enough to calculate simple interest for any set of values without being required to make a change in the program. Moreover, if you distribute the EXE file of this program to somebody he would not even be able to make changes in the program. Hence it is a good practice to create a program that is general enough to work for any set of values.

To make the program general the program itself should ask the user to supply the values of **p**, **n** and **r** through the keyboard during execution. This can be achieved using a function called **scanf( )**. This function is a counter-part of the **printf( )** function. **printf( )** outputs the values to the screen whereas **scanf( )** receives them from the keyboard. This is illustrated in the program shown below.

```
/* Calculation of simple interest */
/* Author gekay Date 25/05/2004 */
main( )
{
    int  p, n ;
    float  r, si ;
    printf ( "Enter values of p, n, r" ) ;
    scanf ( "%d %d %f", &p, &n, &r ) ;

    si = p * n * r / 100 ;
    printf ( "%f" , si ) ;
}
```

The first **printf( )** outputs the message 'Enter values of p, n, r' on the screen. Here we have not used any expression in **printf( )** which means that using expressions in **printf( )** is optional.

Note that the ampersand (**&**) before the variables in the **scanf( )** function is a must. **&** is an 'Address of' operator. It gives the location number used by the variable in memory. When we say **&a**, we are telling **scanf( )** at which memory location should it store the value supplied by the user from the keyboard. The detailed working of the **&** operator would be taken up in Chapter 5.

Note that a blank, a tab or a new line must separate the values supplied to **scanf( )**. Note that a blank is creating using a spacebar, tab using the Tab key and new line using the Enter key. This is shown below:

Ex.: The three values separated by blank

```
1000 5 15.5
```

Ex.: The three values separated by tab.

```
1000	5	15.5
```

Ex.: The three values separated by newline.

```
1000
5
15.5
```

So much for the tips. How about another program to give you a feel of things...

```
/* Just for fun. Author: Bozo */
main( )
{
    int  num ;

    printf ( "Enter a number" ) ;
```

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

    printf ( "Now I am letting you on a secret..." ) ;
    printf ( "You have just entered the number %d", num ) ;
}
```

C Instructions

Now that we have written a few programs let us look at the instructions that we used in these programs. There are basically three types of instructions in C:

(a) Type Declaration Instruction
(b) Arithmetic Instruction
(c) Control Instruction

The purpose of each of these instructions is given below:

(a) Type declaration instruction – To declare the type of variables used in a C program.

(b) Arithmetic instruction – To perform arithmetic operations between constants and variables.

(c) Control instruction – To control the sequence of execution of various statements in a C program.

Since, the elementary C programs would usually contain only the type declaration and the arithmetic instructions; we would discuss only these two instructions at this stage. The other types of instructions would be discussed in detail in the subsequent chapters.

Type Declaration Instruction

This instruction is used to declare the type of variables being used in the program. Any variable used in the program must be declared before using it in any statement. The type declaration statement is written at the beginning of **main( )** function.

```
Ex.: int  bas ;
     float  rs, grosssal ;
     char  name, code ;
```

There are several subtle variations of the type declaration instruction. These are discussed below:

(a) While declaring the type of variable we can also initialize it as shown below.

```
int i = 10, j = 25 ;
float a = 1.5, b = 1.99 + 2.4 * 1.44 ;
```

(b) The order in which we define the variables is sometimes important sometimes not. For example,

```
int i = 10, j = 25 ;
```

is same as

```
int j = 25, j = 10 ;
```

However,

```
float a = 1.5, b = a + 3.1 ;
```

is alright, but

```
float  b = a + 3.1, a = 1.5 ;
```

is not. This is because here we are trying to use a even before defining it.

(c) The following statements would work

```
int a, b, c, d ;
a = b = c = 10 ;
```

However, the following statement would not work

```
int a = b = c = d = 10 ;
```

Once again we are trying to use **b** (to assign to **a**) before defining it.

Arithmetic Instruction

A C arithmetic instruction consists of a variable name on the left hand side of = and variable names & constants on the right hand side of =. The variables and constants appearing on the right hand side of = are connected by arithmetic operators like +, -, *, and /.

```
Ex.: int  ad ;
     float  kot, deta, alpha, beta, gamma ;
     ad = 3200 ;
     kot = 0.0056 ;
     deta = alpha * beta / gamma + 3.2 * 2 / 5 ;
```

Here,

*, /, -, + are the arithmetic operators.
= is the assignment operator.
2, 5 and 3200 are integer constants.
3.2 and 0.0056 are real constants.
ad is an integer variable.
kot, deta, alpha, beta, gamma are real variables.

The variables and constants together are called 'operands' that are operated upon by the 'arithmetic operators' and the result is assigned, using the assignment operator, to the variable on left-hand side.

A C arithmetic statement could be of three types. These are as follows:

(a) Integer mode arithmetic statement - This is an arithmetic statement in which all operands are either integer variables or integer constants.

```
Ex.: int  i, king, issac, noteit ;
     i = i + 1 ;
     king = issac * 234 + noteit - 7689 ;
```

(b) Real mode arithmetic statement - This is an arithmetic statement in which all operands are either real constants or real variables.

```
Ex.: float  qbee, antink, si, prin, anoy, roi ;
     qbee = antink + 23.123 / 4.5 * 0.3442 ;
     si = prin * anoy * roi / 100.0 ;
```

(c) Mixed mode arithmetic statement - This is an arithmetic statement in which some of the operands are integers and some of the operands are real.

```
Ex.: float  si, prin, anoy, roi, avg ;
     int  a, b, c, num ;
     si = prin * anoy * roi / 100.0 ;
     avg = ( a + b + c + num ) / 4 ;
```

It is very important to understand how the execution of an arithmetic statement takes place. Firstly, the right hand side is evaluated using constants and the numerical values stored in the variable names. This value is then assigned to the variable on the left-hand side.

Though Arithmetic instructions look simple to use one often commits mistakes in writing them. Let us take a closer look at these statements. Note the following points carefully.

(a) C allows only one variable on left-hand side of =. That is, **z = k * l** is legal, whereas **k * l = z** is illegal.

(b) In addition to the division operator C also provides a modular division operator. This operator returns the remainder on dividing one integer with another. Thus the expression 10 / 2 yields 5, whereas, 10 % 2 yields 0. Note that the modulus operator (**%**) cannot be applied on a float. Also note that on using % the sign of the remainder is always same as the sign of the numerator. Thus –5 % 2 yields –1, whereas, 5 % -2 yields 1.

(c) An arithmetic instruction is often used for storing character constants in character variables.

```
char  a, b, d ;
a = 'F' ;
b = 'G' ;
d = '+' ;
```

When we do this the ASCII values of the characters are stored in the variables. ASCII values are used to represent any character in memory. The ASCII values of 'F' and 'G' are 70 and 71 (refer the ASCII Table in Appendix E).

(d) Arithmetic operations can be performed on **ints**, **floats** and **chars**.

Thus the statements,

```
char  x, y ;
int  z ;
x = 'a' ;
y = 'b' ;
z = x + y ;
```

are perfectly valid, since the addition is performed on the ASCII values of the characters and not on characters themselves. The ASCII values of 'a' and 'b' are 97 and 98, and hence can definitely be added.

(e) No operator is assumed to be present. It must be written explicitly. In the following example, the multiplication operator after b must be explicitly written.

```
a = c.d.b(xy)               usual arithmetic statement
b = c * d * b * ( x * y )   C statement
```

(f) Unlike other high level languages, there is no operator for performing exponentiation operation. Thus following statements are invalid.

```
a = 3 ** 2 ;
b = 3 ^ 2 ;
```

If we want to do the exponentiation we can get it done this way:

```
#include <math.h>
main( )
{
    int a ;
    a = pow ( 3, 2 ) ;
    printf ( "%d", a ) ;
}
```

Here **pow( )** function is a standard library function. It is being used to raise 3 to the power of 2. **#include <math.h>** is a preprocessor directive. It is being used here to ensure that the **pow( )** function works correctly. We would learn more about standard library functions in Chapter 5 and about preprocessor in Chapter 7.

Integer and Float Conversions

In order to effectively develop C programs, it will be necessary to understand the rules that are used for the implicit conversion of floating point and integer values in C. These are mentioned below. Note them carefully.

(a) An arithmetic operation between an integer and integer always yields an integer result.

(b) An operation between a real and real always yields a real result.

(c) An operation between an integer and real always yields a real result. In this operation the integer is first promoted to a real and then the operation is performed. Hence the result is real.

I think a few practical examples shown in the following figure would put the issue beyond doubt.

Operation	Result	Operation	Result
5 / 2	2	2 / 5	0
5.0 / 2	2.5	2.0 / 5	0.4
5 / 2.0	2.5	2 / 5.0	0.4
5.0 / 2.0	2.5	2.0 / 5.0	0.4

Figure 1.6

Type Conversion in Assignments

It may so happen that the type of the expression and the type of the variable on the left-hand side of the assignment operator may not be same. In such a case the value of the expression is promoted or

demoted depending on the type of the variable on left-hand side of =.

For example, consider the following assignment statements.

```
int i ;
float b ;
i = 3.5 ;
b = 30 ;
```

Here in the first assignment statement though the expression's value is a **float** (3.5) it cannot be stored in **i** since it is an **int**. In such a case the **float** is demoted to an **int** and then its value is stored. Hence what gets stored in **i** is 3. Exactly opposite happens in the next statement. Here, 30 is promoted to 30.000000 and then stored in **b**, since **b** being a **float** variable cannot hold anything except a **float** value.

Instead of a simple expression used in the above examples if a complex expression occurs, still the same rules apply. For example, consider the following program fragment.

```
float a, b, c ;
int s ;
s = a * b * c / 100 + 32 / 4 - 3 * 1.1 ;
```

Here, in the assignment statement some operands are **ints** whereas others are **floats**. As we know, during evaluation of the expression the **ints** would be promoted to **floats** and the result of the expression would be a **float**. But when this **float** value is assigned to **s** it is again demoted to an **int** and then stored in **s**.

Observe the results of the arithmetic statements shown in Figure 1.7. It has been assumed that **k** is an integer variable and **a** is a real variable.

Arithmetic Instruction	Result	Arithmetic Instruction	Result
k = 2 / 9	0	a = 2 / 9	0.0
k = 2.0 / 9	0	a = 2.0 / 9	0.2222
k = 2 / 9.0	0	a = 2 / 9.0	0.2222
k = 2.0 / 9.0	0	a = 2.0 / 9.0	0.2222
k = 9 / 2	4	a = 9 / 2	4.0
k = 9.0 / 2	4	a = 9.0 / 2	4.5
k = 9 / 2.0	4	a = 9 / 2.0	4.5
k = 9.0 / 2.0	4	a = 9.0 / 2.0	4.5

Figure 1.7

Note that though the following statements give the same result, 0, the results are obtained differently.

```
k = 2 / 9 ;
k = 2.0 / 9 ;
```

In the first statement, since both 2 and 9 are integers, the result is an integer, i.e. 0. This 0 is then assigned to **k**. In the second statement 9 is promoted to 9.0 and then the division is performed. Division yields 0.222222. However, this cannot be stored in **k**, **k** being an **int**. Hence it gets demoted to 0 and then stored in **k**.

Hierarchy of Operations

While executing an arithmetic statement, which has two or more operators, we may have some problems as to how exactly does it get executed. For example, does the expression 2 * x - 3 * y correspond to (2x)-(3y) or to 2(x-3y)? Similarly, does A / B * C correspond to A / (B * C) or to (A / B) * C? To answer these questions satisfactorily one has to understand the 'hierarchy' of operations. The priority or precedence in which the operations in

an arithmetic statement are performed is called the hierarchy of operations. The hierarchy of commonly used operators is shown in Figure 1.8.

Priority	Operators	Description
1st	* / %	multiplication, division, modular division
2nd	+ -	addition, subtraction
3rd	=	assignment

Figure 1.8

Now a few tips about usage of operators in general.

(a) Within parentheses the same hierarchy as mentioned in Figure 1.11 is operative. Also, if there are more than one set of parentheses, the operations within the innermost parentheses would be performed first, followed by the operations within the second innermost pair and so on.

(b) We must always remember to use pairs of parentheses. A careless imbalance of the right and left parentheses is a common error. Best way to avoid this error is to type ( ) and then type an expression inside it.

A few examples would clarify the issue further.

Example 1.1: Determine the hierarchy of operations and evaluate the following expression:

```
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8
```

Stepwise evaluation of this expression is shown below:

```
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8
```

```
i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8          operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8              operation: /
i = 1 + 1+ 8 - 2 + 5 / 8                   operation: /
i = 1 + 1 + 8 - 2 + 0                      operation: /
i = 2 + 8 - 2 + 0                          operation: +
i = 10 - 2 + 0                             operation: +
i = 8 + 0                                  operation : -
i = 8                                      operation: +
```

Note that 6 / 4 gives 1 and not 1.5. This so happens because 6 and 4 both are integers and therefore would evaluate to only an integer constant. Similarly 5 / 8 evaluates to zero, since 5 and 8 are integer constants and hence must return an integer value.

Example 1.2: Determine the hierarchy of operations and evaluate the following expression:

```
kk = 3 / 2 * 4 + 3 / 8 + 3
```

Stepwise evaluation of this expression is shown below:

```
kk = 3 / 2 * 4 + 3 / 8 + 3
kk = 1 * 4 + 3 / 8 + 3                     operation: /
kk = 4 + 3 / 8 + 3                         operation: *
kk = 4 + 0 + 3                             operation: /
kk = 4 + 3                                 operation: +
kk = 7                                     operation: +
```

Note that 3 / 8 gives zero, again for the same reason mentioned in the previous example.

All operators in C are ranked according to their precedence. And mind you there are as many as 45 odd operators in C, and these can affect the evaluation of an expression in subtle and unexpected ways if we aren't careful. Unfortunately, there are no simple rules that one can follow, such as "BODMAS" that tells algebra students in which order does an expression evaluate. We have not

encountered many out of these 45 operators, so we won't pursue the subject of precedence any further here. However, it can be realized at this stage that it would be almost impossible to remember the precedence of all these operators. So a full-fledged list of all operators and their precedence is given in Appendix A. This may sound daunting, but when its contents are absorbed in small bites, it becomes more palatable.

So far we have seen how the computer evaluates an arithmetic statement written in C. But our knowledge would be incomplete unless we know how to convert a general arithmetic statement to a C statement. C can handle any complex expression with ease. Some of the examples of C expressions are shown in Figure 1.9.

Algebric Expression	C Expression
$a \times b - c \times d$	a * b - c * d
$(m + n)(a + b)$	(m + n) * (a + b)
$3x2 + 2x + 5$	3 * x * x + 2 * x + 5
$\dfrac{a+b+c}{d+e}$	(a + b + c) / (d + e)
$\left[\dfrac{2BY}{d+1} - \dfrac{x}{3(z+y)}\right]$	2 *.b * y / (d + 1) - x / 3 * (z + y)

Figure 1.9

Associativity of Operators

When an expression contains two operators of equal priority the tie between them is settled using the associativity of the operators. Associativity can be of two types—Left to Right or Right to Left. Left to Right associativity means that the left operand must be

unambiguous. Unambiguous in what sense? It must not be involved in evaluation of any other sub-expression. Similarly, in case of Right to Left associativity the right operand must be unambiguous. Let us understand this with an example.

Consider the expression

```
a = 3 / 2 * 5 ;
```

Here there is a tie between operators of same priority, that is between / and *. This tie is settled using the associativity of / and *. But both enjoy Left to Right associativity. Figure 1.10 shows for each operator which operand is unambiguous and which is not.

Operator	Left	Right	Remark
/	3	2 or 2 * 5	Left operand is unambiguous, Right is not
*	3 / 2 or 2	5	Right operand is unambiguous, Left is not

Figure 1.10

Since both / and * have L to R associativity and only / has unambiguous left operand (necessary condition for L to R associativity) it is performed earlier.

Consider one more expression

```
a = b = 3 ;
```

Here both assignment operators have the same priority and same associativity (Right to Left). Figure 1.11 shows for each operator which operand is unambiguous and which is not.

Operator	Left	Right	Remark
=	a	b or b = 3	Left operand is unambiguous, Right is not
=	b or a = b	3	Right operand is unambiguous, Left is not

Figure 1.11

Since both = have R to L associativity and only the second = has unambiguous right operand (necessary condition for R to L associativity) the second = is performed earlier.

Consider yet another expression

```
z = a * b + c / d ;
```

Here * and / enjoys same priority and same associativity (Left to Right). Figure 1.12 shows for each operator which operand is unambiguous and which is not.

Operator	Left	Right	Remark
*	a	b	Both operands are unambiguous
/	c	d	Both operands are unambiguous

Figure 1.12

Here since left operands for both operators are unambiguous Compiler is free to perform * or / operation as per its convenience

since no matter which is performed earlier the result would be same.

Appendix A gives the associativity of all the operators available in C.

Control Instructions in C

As the name suggests the 'Control Instructions' enable us to specify the order in which the various instructions in a program are to be executed by the computer. In other words the control instructions determine the 'flow of control' in a program. There are four types of control instructions in C. They are:

(a) Sequence Control Instruction
(b) Selection or Decision Control Instruction
(c) Repetition or Loop Control Instruction
(d) Case Control Instruction

The Sequence control instruction ensures that the instructions are executed in the same order in which they appear in the program. Decision and Case control instructions allow the computer to take a decision as to which instruction is to be executed next. The Loop control instruction helps computer to execute a group of statements repeatedly. In the following chapters we are going to learn these instructions in detail. Try your hand at the Exercise presented on the following pages before proceeding to the next chapter, which discusses the decision control instruction.

Summary

(a) The three primary constants and variable types in C are integer, float and character.
(b) A variable name can be of maximum 31 characters.
(c) Do not use a keyword as a variable name.

(d) An expression may contain any sequence of constants, variables and operators.

(e) Operators having equal precedence are evaluated using associativity.

(f) Left to right associativity means that the left operand of a operator must be unambiguous whereas right to left associativity means that the right operand of a operator must be unambiguous.

(g) Input/output in C can be achieved using **scanf( )** and **printf( )** functions.

Exercise

[A] Which of the following are invalid variable names and why?

BASICSALARY	_basic	basic-hra
#MEAN	group.	422
population in 2006	over time	mindovermatter
FLOAT	hELLO	queue.
team'svictory	Plot # 3	2015_DDay

[B] Point out the errors, if any, in the following C statements:

(a) int = 314.562 * 150 ;

(b) name = 'Ajay' ;

(c) varchar = '3' ;

(d) 3.14 * r * r * h = vol_of_cyl ;

(e) k = (a * b) (c + (2.5a + b) (d + e) ;

(f) m_inst = rate of interest * amount in rs ;

(g) si = principal * rateofinterest * numberofyears / 100 ;

(h) area = 3.14 * r ** 2 ;

(i) volume = 3.14 * r ^ 2 * h ;

(j) k = ((a * b) + c) (2.5 * a + b) ;

(k) a = b = 3 = 4 ;

(l) count = count + 1 ;

(m) date = '2 Mar 04' ;

[C] Evaluate the following expressions and show their hierarchy.

(a) g = big / 2 + big * 4 / big - big + abc / 3 ;
(abc = 2.5, big = 2, assume **g** to be a float)

(b) on = ink * act / 2 + 3 / 2 * act + 2 + tig ;
(ink = 4, act = 1, tig = 3.2, assume **on** to be an int)

(c) s = qui * add / 4 - 6 / 2 + 2 / 3 * 6 / god ;
(qui = 4, add = 2, god = 2, assume **s** to be an int)

(d) s = 1 / 3 * a / 4 - 6 / 2 + 2 / 3 * 6 / g ;
(a = 4, g = 3, assume **s** to be an int)

[D] Fill the following table for the expressions given below and then evaluate the result. A sample entry has been filled in the table for expression (a).

Operator	Left	Right	Remark
/	10	5 or 5 / 2 / 1	Left operand is unambiguous, Right is not
..	..	..	..

(a) g = 10 / 5 /2 / 1 ;

(b) b = 3 / 2 + 5 * 4 / 3 ;

(c) a = b = c = 3 + 4 ;

[E] Convert the following equations into corresponding C statements.

(a) $$Z = \frac{8.8 (a + b) 2 / c - 0.5 + 2 a / (q + r)}{(a + b) * (1 / m)}$$

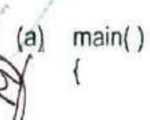
(b) $$X = \frac{- b + (b * b) + 2 \ 4ac}{2a}$$

(c) $$R = \frac{2v + 6.22 \ (c + d)}{g + v}$$

(d) $$A = \frac{7.7b \ (xy + a) / c - 0.8 + 2b}{(x + a) \ (1 / y)}$$

[F] What would be the output of the following programs:

(a)
```
main( )
{
```



```
    int  i = 2, j = 3, k, l ;
    float  a, b ;
    k = i / j * j ;
    l = j / i * i ;
    a = i / j * j ;
    b = j / i * i ;
    printf( "%d %d %f %f", k, l, a, b ) ;
}
```

(b)
```
main( )
{
    int  a, b ;
    a = -3 - - 3 ;
    b = -3 - - ( - 3 ) ;
    printf ( "a = %d b = %d", a, b ) ;
}
```

(c)
```
main( )
{
    float a = 5,  b = 2 ;
    int c ;
    c = a % b ;
    printf ( "%d", c ) ;
}
```

(d)
```
main( )
{
    printf ( "nn \n\n nn\n" ) ;
    printf ( "nn /n/n nn/n" ) ;
}
```

(e)
```
main( )
{
    int a, b ;
    printf ( "Enter values of a and b" ) ;
    scanf ( " %d %d ", &a, &b ) ;
    printf ( "a  = %d b = %d", a, b ) ;
}
```

(f)
```
main( )
{
    int p, q ;
    printf ( "Enter values of p and q" ) ;
    scanf ( " %d %d ", p, q ) ;
    printf ( "p = %d q =%d", p, q ) ;
}
```

[G] Pick up the correct alternative for each of the following questions:

(a) C language has been developed by
 (1) Ken Thompson
 (2) Dennis Ritchie
 (3) Peter Norton
 (4) Martin Richards

(b) C can be used on
 (1) Only MS-DOS operating system
 (2) Only Linux operating system
 (3) Only Windows operating system
 (4) All the above

(c) C programs are converted into machine language with the help of
 (1) An Editor
 (2) A compiler
 (3) An operating system
 (4) None of the above

(d) The real constant in C can be expressed in which of the following forms
 (1) Fractional form only
 (2) Exponential form only
 (3) ASCII form only

(4) Both fractional and exponential forms

(e) A character variable can at a time store
  (1) 1 character
  (2) 8 characters
  (3) 254 characters
  (4) None of the above

(f) The statement **char ch = 'Z'** would store in **ch**
  (1) The character Z
  (2) ASCII value of Z
  (3) Z along with the single inverted commas
  (4) Both (1) and (2)

(g) Which of the following is NOT a character constant
  (1) 'Thank You'
  (2) 'Enter values of P, N, R'
  (3) '23.56E-03'
  (4) All the above

(h) The maximum value that an integer constant can have is
  (1) -32767
  (2) 32767
  (3) 1.7014e+38
  (4) –1.7014e+38

(i) A C variable cannot start with
  (1) An alphabet
  (2) A number
  (3) A special symbol other than underscore
  (4) Both (2) & (3) above

(j) Which of the following statement is wrong
  (1) mes = 123.56 ;
  (2) con = 'T' * 'A' ;
  (3) this = 'T' * 20 ;
  (4) 3 + a = b ;

(k) Which of the following shows the correct hierarchy of arithmetic operators in C
   (1) **, * or /, + or -
   (2) **, *, /, +, -
   (3) **, /, *, +, -
   (4) / or *, - or +

(l) In b = 6.6 / a + 2 * n ; which operation will be performed first?
   (1) 6.6 / a
   (2) a + 2
   (3) 2 * n
   (4) Depends upon compiler

(m) Which of the following is allowed in a C Arithmetic instruction
   (1) [ ]
   (2) { }
   (3) ( )
   (4) None of the above

(n) Which of the following statements is false
   (1) Each new C instruction has to be written on a separate line
   (2) Usually all C statements are entered in small case letters
   (3) Blank spaces may be inserted between two words in a C statement
   (4) Blank spaces cannot be inserted within a variable name

(o) If a is an integer variable, a = 5 / 2 ; will return a value
   (1) 2.5
   (2) 3
   (3) 2
   (4) 0

(p) The expression, a = 7 / 22 * (3.14 + 2) * 3 / 5 ; evaluates to

(1) 8.28
(2) 6.28
(3) 3.14
(4) 0

(q) The expression, a = 30 * 1000 + 2768 ; evaluates to
(1) 32768
(2) -32768
(3) 113040
(4) 0

(r) The expression x = 4 + 2 % - 8 evaluates to
(1) -6
(2) 6
(3) 4
(4) None of the above

(s) Hierarchy decides which operator
(1) is most important
(2) is used first
(3) is fastest
(4) operates on largest numbers

(t) An integer constant in C must have:
(1) At least one digit
(2) Atleast one decimal point
(3) A comma along with digits
(4) Digits separated by commas

(u) A character variable can never store more than
(1) 32 characters
(2) 8 characters
(3) 254 characters
(4) 1 character

(v) In C a variable cannot contain
(1) Blank spaces

(2) Hyphen
(3) Decimal point
(4) All the above

(w) Which of the following is FALSE in C
(1) Keywords can be used as variable names
(2) Variable names can contain a digit
(3) Variable names do not contain a blank space
(4) Capital letters can be used in variable names

(x) In C, Arithmetic instruction cannot contain
(1) variables
(2) constants
(3) variable names on right side of =
(4) constants on left side of =

(y) Which of the following shows the correct hierarchy of arithmetic operations in C
(1) / + * -
(2) * - / +
(3) + - / *
(4) * / + -

(z) What will be the value of **d** if **d** is a float after the operation **d = 2 / 7.0**?
(1) 0
(2) 0.2857
(3) Cannot be determined
(4) None of the above

[H] Write C programs for the following:

(a) Ramesh's basic salary is input through the keyboard. His dearness allowance is 40% of basic salary, and house rent allowance is 20% of basic salary. Write a program to calculate his gross salary.

(b) The distance between two cities (in km.) is input through the keyboard. Write a program to convert and print this distance in meters, feet, inches and centimeters.

(c) If the marks obtained by a student in five different subjects are input through the keyboard, find out the aggregate marks and percentage marks obtained by the student. Assume that the maximum marks that can be obtained by a student in each subject is 100.

(d) Temperature of a city in Fahrenheit degrees is input through the keyboard. Write a program to convert this temperature into Centigrade degrees.

(e) The length & breadth of a rectangle and radius of a circle are input through the keyboard. Write a program to calculate the area & perimeter of the rectangle, and the area & circumference of the circle.

(f) Two numbers are input through the keyboard into two locations C and D. Write a program to interchange the contents of C and D.

(g) If a five-digit number is input through the keyboard, write a program to calculate the sum of its digits.

(Hint: Use the modulus operator '%')

(h) If a five-digit number is input through the keyboard, write a program to reverse the number.

(i) If a four-digit number is input through the keyboard, write a program to obtain the sum of the first and last digit of this number.

(j) In a town, the percentage of men is 52. The percentage of total literacy is 48. If total percentage of literate men is 35 of the total population, write a program to find the total number

of illiterate men and women if the population of the town is 80,000.

(k) A cashier has currency notes of denominations 10, 50 and 100. If the amount to be withdrawn is input through the keyboard in hundreds, find the total number of currency notes of each denomination the cashier will have to give to the withdrawer.

(l) If the total selling price of 15 items and the total profit earned on them is input through the keyboard, write a program to find the cost price of one item.

(m) If a five-digit number is input through the keyboard, write a program to print a new number by adding one to each of its digits. For example if the number that is input is 12391 then the output should be displayed as 23402.

2 The Decision Control Structure

- Decisions! Decisions!
- The *if* Statement
  - The Real Thing
  - Multiple Statements within *if*
- The *if-else* Statement
  - Nested *if-elses*
  - Forms of *if*
- Use of Logical Operators
  - The *else if* Clause
  - The ! Operator
  - Hierarchy of Operators Revisited
- A Word of Caution
- The Conditional Operators
- Summary
- Exercise

We all need to alter our actions in the face of changing circumstances. If the weather is fine, then I will go for a stroll. If the highway is busy I would take a diversion. If the pitch takes spin, we would win the match. If she says no, I would look elsewhere. If you like this book, I would write the next edition. You can notice that all these decisions depend on some condition being met.

C language too must be able to perform different sets of actions depending on the circumstances. In fact this is what makes it worth its salt. C has three major decision making instructions—the **if** statement, the **if-else** statement, and the **switch** statement. A fourth, somewhat less important structure is the one that uses conditional operators. In this chapter we will explore all these ways (except **switch**, which has a separate chapter devoted to it, later) in which a C program can react to changing circumstances.

Decisions! Decisions!

In the programs written in Chapter 1 we have used sequence control structure in which the various steps are executed sequentially, i.e. in the same order in which they appear in the program. In fact to execute the instructions sequentially, we don't have to do anything at all. By default the instructions in a program are executed sequentially. However, in serious programming situations, seldom do we want the instructions to be executed sequentially. Many a times, we want a set of instructions to be executed in one situation, and an entirely different set of instructions to be executed in another situation. This kind of situation is dealt in C programs using a decision control instruction. As mentioned earlier, a decision control instruction can be implemented in C using:

(a) The **if** statement
(b) The **if-else** statement
(c) The conditional operators

Now let us learn each of these and their variations in turn.

The *if* Statement

Like most languages, C uses the keyword **if** to implement the decision control instruction. The general form of **if** statement looks like this:

```
if ( this condition is true )
    execute this statement ;
```

The keyword **if** tells the compiler that what follows is a decision control instruction. The condition following the keyword **if** is always enclosed within a pair of parentheses. If the condition, whatever it is, is true, then the statement is executed. If the condition is not true then the statement is not executed; instead the program skips past it. But how do we express the condition itself in C? And how do we evaluate its truth or falsity? As a general rule, we express a condition using C's 'relational' operators. The relational operators allow us to compare two values to see whether they are equal to each other, unequal, or whether one is greater than the other. Here's how they look and how they are evaluated in C.

this expression	**is true if**
x == y	x is equal to y
x != y	x is not equal to y
x < y	x is less than y
x > y	x is greater than y
x <= y	x is less than or equal to y
x >= y	x is greater than or equal to y

Figure 2.1

The relational operators should be familiar to you except for the equality operator == and the inequality operator !=. Note that = is used for assignment, whereas, == is used for comparison of two quantities. Here is a simple program, which demonstrates the use of **if** and the relational operators.

```
/* Demonstration of if statement */
main( )
{
    int  num ;

    printf ( "Enter a number less than 10 " ) ;
    scanf ( "%d", &num ) ;

    if ( num <= 10 )
        printf ( "What an obedient servant you are !" ) ;
}
```

On execution of this program, if you type a number less than or equal to 10, you get a message on the screen through **printf( )**. If you type some other number the program doesn't do anything. The following flowchart would help you understand the flow of control in the program.

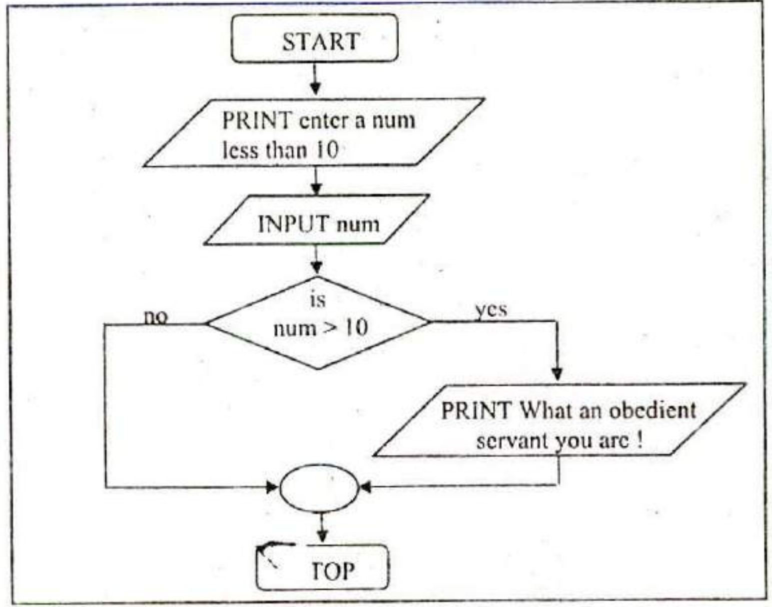


Figure 2.2

To make you comfortable with the decision control instruction one more example has been given below. Study it carefully before reading further. To help you understand it easily, the program is accompanied by an appropriate flowchart.

Example 2.1: While purchasing certain items, a discount of 10% is offered if the quantity purchased is more than 1000. If quantity and price per item are input through the keyboard, write a program to calculate the total expenses.

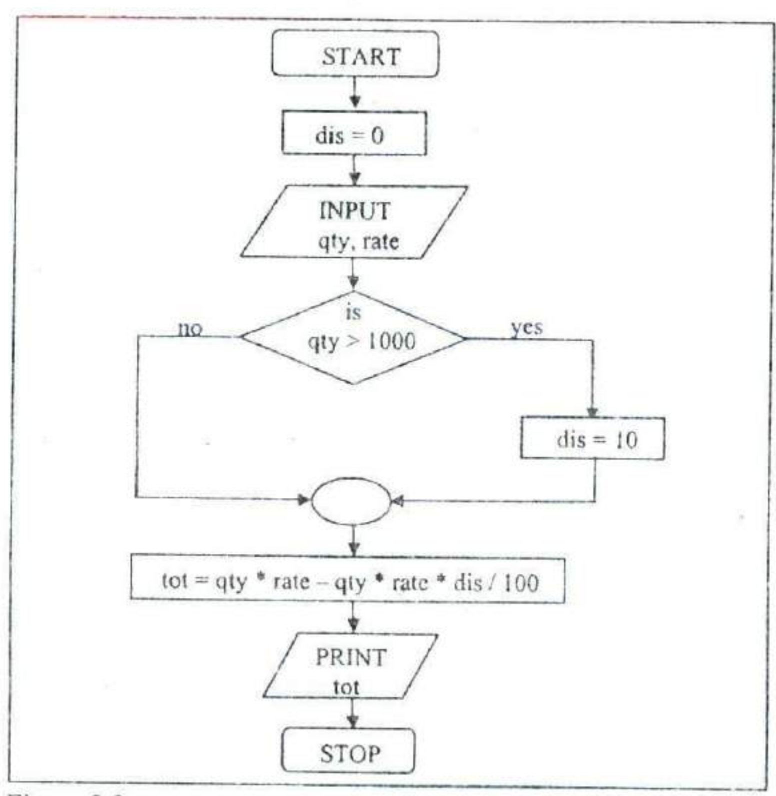


Figure 2.3

```
/* Calculation of total expenses */
main( )
{
    int  qty, dis = 0 ;
    float  rate, tot ;
    printf ( "Enter quantity and rate " ) ;
    scanf ( "%d %f", &qty, &rate) ;

    if ( qty > 1000 )
        dis = 10 ;
```

```
    tot = ( qty * rate ) - ( qty * rate * dis / 100 ) ;
    printf ( "Total expenses = Rs. %f", tot ) ;
}
```

Here is some sample interaction with the program.

```
Enter quantity and rate 1200 15.50
Total expenses = Rs. 16740.000000

Enter quantity and rate 200 15.50
Total expenses = Rs. 3100.000000
```

In the first run of the program, the condition evaluates to true, as 1200 (value of **qty**) is greater than 1000. Therefore, the variable **dis**, which was earlier set to 0, now gets a new value 10. Using this new value total expenses are calculated and printed.

In the second run the condition evaluates to false, as 200 (the value of **qty**) isn't greater than 1000. Thus, **dis**, which is earlier set to 0, remains 0, and hence the expression after the minus sign evaluates to zero, thereby offering no discount.

Is the statement **dis = 0** necessary? The answer is yes, since in C, a variable if not specifically initialized contains some unpredictable value (garbage value).

The Real Thing

We mentioned earlier that the general form of the if statement is as follows

```
if ( condition )
    statement ;
```

Truly speaking the general form is as follows:

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

Here the expression can be any valid expression including a relational expression. We can even use arithmetic expressions in the **if** statement. For example all the following **if** statements are valid

```
if ( 3 + 2 % 5 )
    printf ( "This works" ) ;

if ( a = 10 )
    printf ( "Even this works" ) ;

if ( -5 )
    printf ( "Surprisingly even this works" ) ;
```

Note that in C a non-zero value is considered to be true, whereas a 0 is considered to be false. In the first **if**, the expression evaluates to **5** and since **5** is non-zero it is considered to be true. Hence the **printf( )** gets executed.

In the second **if**, 10 gets assigned to **a** so the **if** is now reduced to **if (a)** or **if (10)**. Since 10 is non-zero, it is true hence again **printf( )** goes to work.

In the third **if**, -5 is a non-zero number, hence true. So again **printf( )** goes to work. In place of -5 even if a float like 3.14 were used it would be considered to be true. So the issue is not whether the number is integer or float, or whether it is positive or negative. Issue is whether it is zero or non-zero.

Multiple Statements within *if*

It may so happen that in a program we want more than one statement to be executed if the expression following **if** is satisfied. If such multiple statements are to be executed then they must be

placed within a pair of braces as illustrated in the following example.

Example 2.2: The current year and the year in which the employee joined the organization are entered through the keyboard. If the number of years for which the employee has served the organization is greater than 3 then a bonus of Rs. 2500/- is given to the employee. If the years of service are not greater than 3, then the program should do nothing.

```
/* Calculation of bonus */
main( )
{
    int  bonus, cy, yoj, yr_of_ser ;

    printf ( "Enter current year and year of joining " ) ;
    scanf ( "%d %d", &cy, &yoj ) ;

    yr_of_ser = cy - yoj ;

    if ( yr_of_ser > 3 )
    {
        bonus = 2500 ;
        printf ( "Bonus = Rs. %d", bonus ) ;
    }
}
```

Observe that here the two statements to be executed on satisfaction of the condition have been enclosed within a pair of braces. If a pair of braces is not used then the C compiler assumes that the programmer wants only the immediately next statement after the **if** to be executed on satisfaction of the condition. In other words we can say that the default scope of the **if** statement is the immediately next statement after it.

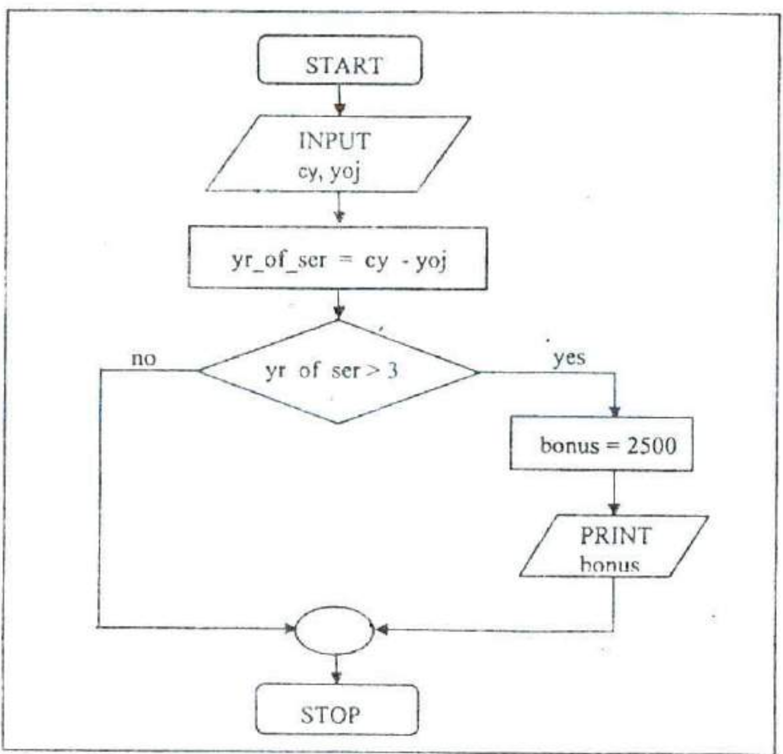


Figure 2.4

The *if-else* Statement

The **if** statement by itself will execute a single statement, or a group of statements, when the expression following **if** evaluates to true. It does nothing when the expression evaluates to false. Can we execute one group of statements if the expression evaluates to true and another group of statements if the expression evaluates to false? Of course! This is what is the purpose of the **else** statement that is demonstrated in the following example:

Example 2.3: In a company an employee is paid as under:

If his basic salary is less than Rs. 1500, then HRA = 10% of basic salary and DA = 90% of basic salary. If his salary is either equal to or above Rs. 1500, then HRA = Rs. 500 and DA = 98% of basic salary. If the employee's salary is input through the keyboard write a program to find his gross salary.

```
/* Calculation of gross salary */
main( )
{
    float  bs, gs, da, hra ;

    printf ( "Enter basic salary " ) ;
    scanf ( "%f", &bs ) ;

    if ( bs < 1500 )
    {
        hra = bs * 10 / 100 ;
        da = bs * 90 / 100 ;
    }
    else
    {
        hra = 500 ;
        da = bs * 98 / 100 ;
    }

    gs = bs + hra + da ;
    printf ( "gross salary = Rs. %f", gs ) ;
}
```

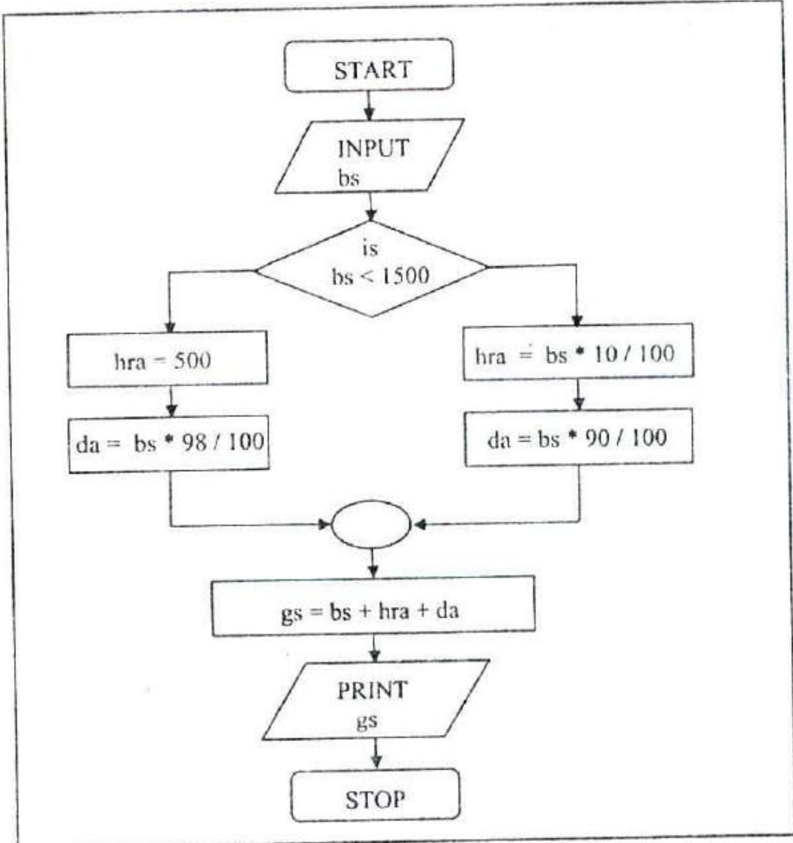


Figure 2.5

A few points worth noting...

(a) The group of statements after the **if** upto and not including the **else** is called an 'if block'. Similarly, the statements after the **else** form the 'else block'.

(b) Notice that the **else** is written exactly below the **if**. The statements in the if block and those in the else block have been indented to the right. This formatting convention is

followed throughout the book to enable you to understand the working of the program better.

(c) Had there been only one statement to be executed in the if block and only one statement in the else block we could have dropped the pair of braces.

(d) As with the **if** statement, the default scope of **else** is also the statement immediately after the **else**. To override this default scope a pair of braces as shown in the above example must be used.

Nested *if-elses*

It is perfectly all right if we write an entire **if-else** construct within either the body of the **if** statement or the body of an **else** statement. This is called 'nesting'of **ifs**. This is shown in the following program.

```
/* A quick demo of nested if-else */
main( )
{
    int  i ;

    printf ( "Enter either 1 or 2 " ) ;
    scanf ( "%d", &i ) ;

    if ( i == 1 )
        printf ( "You would go to heaven !" ) ;
    else
    {
        if ( i == 2 )
            printf ( "Hell was created with you in mind" ) ;
        else
            printf ( "How about mother earth !" ) ;
    }
}
```

Note that the second **if-else** construct is nested in the first **else** statement. If the condition in the first **if** statement is false, then the condition in the second **if** statement is checked. If it is false as well, then the final **else** statement is executed.

You can see in the program how each time a **if-else** construct is nested within another **if-else** construct, it is also indented to add clarity to the program. Inculcate this habit of indentation, otherwise you would end up writing programs which nobody (you included) can understand easily at a later date.

In the above program an **if-else** occurs within the **else** block of the first **if** statement. Similarly, in some other program an **if-else** may occur in the **if** block as well. There is no limit on how deeply the **if**s and the **else**s can be nested.

Forms of *if*

The **if** statement can take any of the following forms:

```
(a)  if ( condition )
         do this ;

(b)  if ( condition )
     {
         do this ;
         and this ;
     }

(c)  if ( condition )
         do this ;
     else
         do this ;
(d)  if ( condition )
     {
         do this ;
```

```
        and this ;
    }
    else
    {
        do this ;
        and this ;
    }
```

```
(e) if ( condition )
        do this ;
    else
    {
        if ( condition )
            do this ;
        else
        {
            do this ;
            and this ;
        }
    }
```

```
(f) if ( condition )
    {
        if ( condition )
            do this ;
        else
        {
            do this ;
            and this ;
        }
    }
    else
        do this ;
```

Use of Logical Operators

C allows usage of three logical operators, namely, &&, || and !. These are to be read as 'AND' 'OR' and 'NOT' respectively.

There are several things to note about these logical operators. Most obviously, two of them are composed of double symbols: || and **&&**. Don't use the single symbol | and **&**. These single symbols also have a meaning. They are bitwise operators, which we would examine in Chapter 14.

The first two operators, **&&** and ||, allow two or more conditions to be combined in an **if** statement. Let us see how they are used in a program. Consider the following example.

Example 2.4: The marks obtained by a student in 5 different subjects are input through the keyboard. The student gets a division as per the following rules:

Percentage above or equal to 60 - First division
Percentage between 50 and 59 - Second division
Percentage between 40 and 49 - Third division
Percentage less than 40 - Fail

Write a program to calculate the division obtained by the student.

There are two ways in which we can write a program for this example. These methods are given below.

```
/* Method – I */
main( )
{
    int  m1, m2, m3, m4, m5, per ;

    printf ( "Enter marks in five subjects " ) ;
    scanf ( "%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5 ) ;
    per = ( m1 + m2 + m3 + m4 + m5 ) / 5 ;
```

```
    if ( per >= 60 )
        printf ( "First division ") ;
    else
    {
        if ( per >= 50 )
            printf ( "Second division" ) ;
        else
        {
            if ( per >= 40 )
                printf ( "Third division" ) ;
            else
                printf ( "Fail" ) ;
        }
    }
}
```

This is a straight forward program. Observe that the program uses nested **if-elses**. This leads to three disadvantages:

(a) As the number of conditions go on increasing the level of indentation also goes on increasing. As a result the whole program creeps to the right.
(b) Care needs to be exercised to match the corresponding **ifs** and **elses**.
(c) Care needs to be exercised to match the corresponding pair of braces.

All these three problems can be eliminated by usage of 'Logical operators'. The following program illustrates this.

```
/* Method – II */
main( )
{
    int m1, m2, m3, m4, m5, per ;

    printf ( "Enter marks in five subjects " ) ;
    scanf ( "%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5 ) ;
    per = ( m1 + m2 + m3 + m4 + m5 ) / 5 ;
```

```
    if ( per >= 60 )
        printf ( "First division" ) ;

    if ( ( per >= 50 ) && ( per < 60 ) )
        printf ( "Second division" ) ;

    if ( ( per >= 40 ) && ( per < 50 ) )
        printf ( "Third division" ) ;

    if ( per < 40 )
        printf ( "Fail" ) ;
}
```

As can be seen from the second **if** statement, the **&&** operator is used to combine two conditions. 'Second division' gets printed if both the conditions evaluate to true. If one of the conditions evaluate to false then the whole thing is treated as false.

Two distinct advantages can be cited in favour of this program:

(a) The matching (or do I say mismatching) of the **if**s with their corresponding **else**s gets avoided, since there are no **else**s in this program.

(b) In spite of using several conditions, the program doesn't creep to the right. In the previous program the statements went on creeping to the right. This effect becomes more pronounced as the number of conditions go on increasing. This would make the task of matching the **if**s with their corresponding **else**s and matching of opening and closing braces that much more difficult.

The *else if* Clause

There is one more way in which we can write program for Example 2.4. This involves usage of **else if** blocks as shown below:

```
/* else if ladder demo */
main( )
{
    int  m1, m2, m3, m4, m5, per ;

    per = ( m1+ m2 + m3 +  m4+ m5 ) / per ;

    if ( per >= 60 )
        printf ( "First division" ) ;
    else if ( per >= 50 )
        printf ( "Second division" ) ;
    else if ( per >= 40 )
        printf ( "Third division" ) ;
    else
        printf ( "fail" ) ;
}
```

You can note that this program reduces the indentation of the statements. In this case every **else** is associated with its previous **if**. The last **else** goes to work only if all the conditions fail. Even in **else if** ladder the last **else** is optional.

Note that the **else if** clause is nothing different. It is just a way of rearranging the **else** with the **if** that follows it. This would be evident if you look at the following code:

```
if ( i == 2 )
    printf (  "With you..." ) ;
else
{
    if ( j == 2 )
    printf ( "...All the time" ) ;
}
```

```
if ( i == 2 )
    printf (  "With you..." ) ;
else if ( j == 2 )
    printf ( "...All the time " ) ;
```

Another place where logical operators are useful is when we want to write programs for complicated logics that ultimately boil down

to only two answers. For example, consider the following example:

Example 2.5: A company insures its drivers in the following cases:

- If the driver is married.
- If the driver is unmarried, male & above 30 years of age.
- If the driver is unmarried, female & above 25 years of age.

In all other cases the driver is not insured. If the marital status, sex and age of the driver are the inputs, write a program to determine whether the driver is to be insured or not.

Here after checking a complicated set of instructions the final output of the program would be one of the two—Either the driver should be ensured or the driver should not be ensured. As mentioned above, since these are the only two outcomes this problem can be solved using logical operators. But before we do that let us write a program that does not make use of logical operators.

```
/* Insurance of driver - without using logical operators */
main( )
{
    char  sex, ms ;
    int  age ;

    printf ( "Enter age, sex, marital status " ) ;
    scanf ( "%d %c %c", &age, &sex, &ms ) ;

    if ( ms == 'M' )
        printf ( "Driver is insured" ) ;
    else
    {
        if ( sex == 'M' )
        {
```

```
            if ( age > 30 )
                printf ( "Driver is insured" ) ;
            else
                printf ( "Driver is not insured" ) ;
        }
        else
        {
            if ( age > 25 )
                printf ( "Driver is insured" ) ;
            else
                printf ( "Driver is not insured" ) ;
        }
    }
}
```

From the program it is evident that we are required to match several **if**s and **else**s and several pairs of braces. In a more real-life situation there would be more conditions to check leading to the program creeping to the right. Let us now see how to avoid these problems by using logical operators.

As mentioned above, in this example we expect the answer to be either 'Driver is insured' or 'Driver is not insured'. If we list down all those cases in which the driver is insured, then they would be:

(a) Driver is married.
(b) Driver is an unmarried male above 30 years of age.
(c) Driver is an unmarried female above 25 years of age.

Since all these cases lead to the driver being insured, they can be combined together using **&&** and **||** as shown in the program below:

```
/* Insurance of driver - using logical operators */
main( )
{
    char  sex, ms ;
```

```
    int age ;

    printf ( "Enter age, sex, marital status " ) ;
    scanf ( "%d %c %c" &age, &sex, &ms ) ;

    if ( ( ms == 'M') || ( ms == 'U' && sex == 'M' && age > 30 ) ||
                ( ms == 'U' && sex == 'F' && age > 25 ) )
        printf ( "Driver is insured" ) ;
    else
        printf ( "Driver is not insured" ) ;
}
```

In this program it is important to note that:

- The driver will be insured only if one of the conditions enclosed in parentheses evaluates to true.

- For the second pair of parentheses to evaluate to true, each condition in the parentheses separated by **&&** must evaluate to true.

- Even if one of the conditions in the second parentheses evaluates to false, then the whole of the second parentheses evaluates to false.

- The last two of the above arguments apply to third pair of parentheses as well.

Thus we can conclude that the **&&** and || are useful in the following programming situations:

(a) When it is to be tested whether a value falls within a particular range or not.
(b) When after testing several conditions the outcome is only one of the two answers (This problem is often called yes/no problem).

There can be one more situation other than checking ranges or yes/no problem where you might find logical operators useful. The following program demonstrates it.

Example 2.6: Write a program to calculate the salary as per the following table:

Gender	Years of Service	Qualifications	Salary
Male	>= 10	Post-Graduate	15000
	>= 10	Graduate	10000
	< 10	Post-Graduate	10000
	< 10	Graduate	7000
Female	>= 10	Post-Graduate	12000
	>= 10	Graduate	9000
	< 10	Post-Graduate	10000
	< 10	Graduate	6000

Figure 2.6

```
main( )
{
    char  g ;
    int   yos, qual, sal ;

    printf ( "Enter Gender, Years of Service and
             Qualifications ( 0 = G, 1 = PG ):" ) ;
    scanf ( "%c%d%d", &g, &yos, &qual ) ;

    if ( g == 'm' && yos >= 10 && qual == 1 )
        sal = 15000 ;
    else if ( ( g == 'm' && yos >= 10 && qual == 0 ) ||
        ( g == 'm' && yos < 10 && qual == 1 ) )
        sal = 10000 ;
```

```
    else if ( g == 'm' && yos < 10 && qual == 0 )
        sal = 7000 ;
    else if ( g == 'f' && yos >= 10 && qual == 1 )
        sal = 12000 ;
    else if ( g == 'f' && yos >= 10 && qual == 0 )
        sal = 9000 ;
    else if ( g == 'f' && yos < 10 && qual == 1 )
        sal = 10000 ;
    else if ( g == 'f' && yos < 10 && qual == 0 )
        sal = 6000 ;

    printf ( "\nSalary of Employee = %d", sal ) ;
}
```

The ! Operator

So far we have used only the logical operators **&&** and **||**. The third logical operator is the NOT operator, written as **!**. This operator reverses the result of the expression it operates on. For example, if the expression evaluates to a non-zero value, then applying **!** operator to it results into a 0. Vice versa, if the expression evaluates to zero then on applying **!** operator to it makes it 1, a non-zero value. The final result (after applying **!**) 0 or 1 is considered to be false or true respectively. Here is an example of the NOT operator applied to a relational expression.

```
! ( y < 10 )
```

This means "not **y** less than 10". In other words, if **y** is less than 10, the expression will be false, since **(y < 10)** is true. We can express the same condition as **(y >= 10)**.

The NOT operator is often used to reverse the logical value of a single variable, as in the expression

```
if ( ! flag )
```

This is another way of saying

```
if ( flag == 0 )
```

Does the NOT operator sound confusing? Avoid it if you want, as the same thing can be achieved without using the NOT operator.

Hierarchy of Operators Revisited

Since we have now added the logical operators to the list of operators we know, it is time to review these operators and their priorities. Figure 2.7 summarizes the operators we have seen so far. The higher the position of an operator is in the table, higher is its priority. (A full-fledged precedence table of operators is given in Appendix A.)

Operators	Type
!	Logical NOT
* / %	Arithmetic and modulus
+ -	Arithmetic
< > <= >=	Relational
== !=	Relational
&&	Logical AND
||	Logical OR
=	Assignment

Figure 2.7

A Word of Caution

What will be the output of the following program:

```
main( )
{
    int  i ;

    printf ( "Enter value of i " ) ;
    scanf ( "%d", &i ) ;
    if ( i = 5 )
        printf ( "You entered 5" ) ;
    else
        printf ( "You entered something other than 5" ) ;
}
```

And here is the output of two runs of this program...

```
Enter value of i 200
You entered 5
Enter value of i 9999
You entered 5
```

Surprising? You have entered 200 and 9999, and still you find in either case the output is 'You entered 5'. This is because we have written the condition wrongly. We have used the assignment operator = instead of the relational operator ==. As a result, the condition gets reduced to **if (5)**, irrespective of what you supply as the value of **i**. And remember that in C 'truth' is always non-zero, whereas 'falsity' is always zero. Therefore, **if (5)** always evaluates to true and hence the result.

Another common mistake while using the **if** statement is to write a semicolon (;) after the condition, as shown below:

```
main( )
{
    int  i ;

    printf ( "Enter value of i " ) ;
    scanf ( "%d", &i ) ;
```

```
    if ( i == 5 ) ;
        printf ( "You entered 5" ) ;
}
```

The ; makes the compiler to interpret the statement as if you have written it in following manner:

```
if ( i == 5 )
    ;
printf ( "You entered 5" ) ;
```

Here, if the condition evaluates to true the ; (null statement, which does nothing on execution) gets executed, following which the **printf( )** gets executed. If the condition fails then straightaway the **printf( )** gets executed. Thus, irrespective of whether the condition evaluates to true or false the **printf( )** is bound to get executed. Remember that the compiler would not point out this as an error, since as far as the syntax is concerned nothing has gone wrong, but the logic has certainly gone awry. Moral is, beware of such pitfalls.

The following figure summarizes the working of all the three logical operators.

Operands		**Results**			
x	**y**	**!x**	**!y**	**x && y**	**x || y**
0	0	1	1	0	0
0	non-zero	1	0	0	0
non-zero	0	0	1	0	1
non-zero	non-zero	0	0	1	1

Figure 2.8

The Conditional Operators

The conditional operators ? and : are sometimes called ternary operators since they take three arguments. In fact, they form a kind of foreshortened if-then-else. Their general form is,

```
expression 1 ? expression 2 : expression 3
```

What this expression says is: "if **expression 1** is true (that is, if its value is non-zero), then the value returned will be **expression 2**, otherwise the value returned will be **expression 3**". Let us understand this with the help of a few examples:

(a)
```
int x, y ;
scanf ( "%d", &x ) ;
y = ( x > 5 ? 3 : 4 ) ;
```

This statement will store 3 in **y** if **x** is greater than 5, otherwise it will store 4 in y.

The equivalent **if** statement will be,

```
if ( x > 5 )
    y = 3 ;
else
    y = 4 ;
```

(b)
```
char a ;
int y ;
scanf ( "%c", &a ) ;
y = ( a >= 65 && a <= 90 ? 1 : 0 ) ;
```

Here 1 would be assigned to **y** if **a >=65 && a <=90** evaluates to true, otherwise 0 would be assigned.

The following points may be noted about the conditional operators:

(a) It's not necessary that the conditional operators should be used only in arithmetic statements. This is illustrated in the following examples:

```
Ex.:    int i ;
        scanf ( "%d", &i ) ;
        ( i == 1 ? printf ( "Amit" ) : printf ( "All and sundry" ) ) ;
```

```
Ex.:    char a = 'z' ;
        printf ( "%c" , ( a >= 'a' ? a : '!' ) ) ;
```

(b) The conditional operators can be nested as shown below.

```
int big, a, b, c ;
big = ( a > b ? ( a > c ? 3: 4 ) : ( b > c ? 6: 8 ) ) ;
```

(c) Check out the following conditional expression:

```
a > b ? g = a : g = b ;
```

This will give you an error 'Lvalue Required'. The error can be overcome by enclosing the statement in the **:** part within a pair of parenthesis. This is shown below:

```
a > b ? g = a : ( g = b ) ;
```

In absence of parentheses the compiler believes that **b** is being assigned to the result of the expression to the left of second =. Hence it reports an error.

The limitation of the conditional operators is that after the **?** or after the **:** only one C statement can occur. In practice rarely is this the requirement. Therefore, in serious C programming conditional operators aren't as frequently used as the **if-else**.

Summary

(a) There are three ways for taking decisions in a program. First way is to use the **if-else** statement, second way is to use the

conditional operators and third way is to use the **switch** statement.

(b) The default scope of the **if** statement is only the next statement. So, to execute more than one statement they must be written in a pair of braces.

(c) An **if** block need not always be associated with an **else** block. However, an **else** block is always associated with an **if** statement.

(d) If the outcome of an **if-else** ladder is only one of two answers then the ladder should be replaced either with an **else-if** clause or by logical operators.

(e) **&&** and || are binary operators, whereas, ! is a unary operator.

(f) In C every test expression is evaluated in terms of zero and non-zero values. A zero value is considered to be false and a non-zero value is considered to be true.

(g) Assignment statements used with conditional operators must be enclosed within a pair of parenthesis.

Exercise

if, *if-else*, Nested *if-elses*

[A] What would be the output of the following programs:

(a)
```
main( )
{
    int  a = 300, b, c ;
    if ( a >= 400 )
        b = 300 ;
        c = 200 ;
        printf ( "\n%d %d", b, c ) ;
}
```

(b)
```
main( )
{
    int  a = 500, b, c ;
    if ( a >= 400 )
```

```
        b = 300 ;
        c = 200 ;
        printf ( "\n%d %d", b, c ) ;
    }
```

(c)
```
main( )
{
    int  x = 10, y = 20 ;
    if ( x == y ) ;
        printf ( "\n%d %d", x, y ) ;
}
```

(d)
```
main( )
{
    int  x = 3, y = 5 ;
    if ( x == 3 )
        printf ( "\n%d", x ) ;
    else ;
        printf ( "\n%d", y ) ;
}
```

(e)
```
main( )
{
    int  x = 3 ;
    float  y = 3.0 ;

    if ( x == y )
        printf ( "\nx and y are equal" ) ;
    else
        printf ( "\nx and y are not equal" ) ;
}
```

(f)
```
main( )
{
    int  x = 3, y, z ;
    y = x = 10 ;
    z = x < 10 ;
    printf ( "\nx = %d y = %d z = %d", x, y, z ) ;
}
```

(g)
```
main( )
{
    int k = 35 ;
    printf ( "\n%d %d %d", k == 35, k = 50, k > 40 ) ;
}
```

(h)
```
main( )
{
    int i = 65 ;
    char j = 'A' ;
    if ( i == j )
        printf ( "C is WOW" ) ;
    else
        printf( "C is a headache" ) ;
}
```

(i)
```
main( )
{
    int a = 5, b, c ;
    b = a = 15 ;
    c = a < 15 ;
    printf ( "\na = %d b = %d c = %d", a, b, c ) ;
}
```

(j)
```
main( )
{
    int x = 15 ;
    printf ( "\n%d %d %d", x != 15, x = 20, x < 30 ) ;
}
```

[B] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    float a = 12.25, b = 12.52 ;
    if ( a = b )
        printf ( "\na and b are equal" ) ;
```

(b)

```
! j = 10, k = 12 ;
k >= j )

  {
      k = j ;
      j = k ;
  }
```

(c) ( )

```
'X' < 'x' )
  printf ( "\nascii value of X is smaller than that of x" ) ;
```

(d) n)

```
int x = 10 ;
if ( x >= 2 ) then
     printf ( "\n%d", x ) ;
```

```
ain( )
{
   int x = 10 ;
   if x >= 2
        printf ( "\n%d", x ) ;
}
```

(f)
```
main( )
{
   int x = 10, y = 15 ;
   if ( x % 2 = y % 3 )
```

```
        printf ( "\nCarpathians" ) ;
}
```

(g)
```
main( )
{
    int x = 30 , y = 40 ;
    if ( x == y )
        printf( "x is equal to y" ) ;
    elseif ( x > y )
        printf( "x is greater than y" ) ;
    elseif ( x < y )
        printf( "x is less than y" ) ;
}
```

(h)
```
main( )
{
    int  x = 10 ;
    if ( x >= 2 ) then
        printf ( "\n%d", x ) ;
}
```

(i)
```
main( )
{
    int a, b ;
    scanf ( "%d %d",a, b ) ;
    if ( a > b ) ;
        printf ( "This is a game" ) ;
    else
        printf ( "You have to play it" ) ;
}
```

[C] Attempt the following:

(a) If cost price and selling price of an item is input through the keyboard, write a program to determine whether the seller has made profit or incurred loss. Also determine how much profit he made or loss he incurred.

(b) Any integer is input through the keyboard. Write a program to find out whether it is an odd number or even number.

(c) Any year is input through the keyboard. Write a program to determine whether the year is a leap year or not.

(Hint: Use the % (modulus) operator)

(d) According to the Gregorian calendar, it was Monday on the date 01/01/1900. If any year is input through the keyboard write a program to find out what is the day on 1st January of this year.

(e) A five-digit number is entered through the keyboard. Write a program to obtain the reversed number and to determine whether the original and reversed numbers are equal or not.

(f) If the ages of Ram, Shyam and Ajay are input through the keyboard, write a program to determine the youngest of the three.

(g) Write a program to check whether a triangle is valid or not, when the three angles of the triangle are entered through the keyboard. A triangle is valid if the sum of all the three angles is equal to 180 degrees.

(h) Find the absolute value of a number entered through the keyboard.

(i) Given the length and breadth of a rectangle, write a program to find whether the area of the rectangle is greater than its perimeter. For example, the area of the rectangle with length = 5 and breadth = 4 is greater than its perimeter.

(j) Given three points **(x1, y1)**, **(x2, y2)** and **(x3, y3)**, write a program to check if all the three points fall on one straight line.

(k) Given the coordinates **(x, y)** of a center of a circle and it's radius, write a program which will determine whether a point lies inside the circle, on the circle or outside the circle.

(Hint: Use **sqrt( )** and **pow( )** functions)

(l) Given a point **(x, y)**, write a program to find out if it lies on the x-axis, y-axis or at the origin, viz. (0, 0).

Logical Operators

If a = 10, b = 12, c = 0, find the values of the expressions in the following table:

Expression	**Value**
a != 6 && b > 5 a == 9 || b < 3 ! (a < 10) ! (a > 5 && c) 5 && c != 8 || !c	1

[D] What would be the output of the following programs:

(a)
```
main( )
{
    int  i = 4, z = 12 ;
    if ( i = 5 || z > 50 )
        printf ( "\nDean of students affairs" ) ;
    else
        printf ( "\nDosa" ) ;
}
```

(b)
```
main( )
{
    int  i = 4, z = 12 ;
```

```
    if ( i = 5 && z > 5 )
        printf ( "\nLet us C" ) ;
    else
        printf ( "\nWish C was free !" ) ;
}
```

(c)
```
main( )
{
    int  i = 4, j = -1, k = 0, w, x, y, z ;
    w = i || j || k ;
    x = i && j && k ;
    y = i || j && k ;
    z = i && j || k ;
    printf ( "\nw = %d x = %d y = %d z = %d", w, x, y, z ) ;
}
```

(d)
```
main( )
{
    int  i = 4, j = -1, k = 0, y, z ;
    y = i + 5 && j + 1 || k + 2 ;
    z = i + 5 || j + 1 && k + 2 ;
    printf ( "\ny = %d z = %d", y, z ) ;
}
```

(e)
```
main( )
{
    int  i = -3, j = 3 ;
    if ( !i + !j * 1 )
        printf ( "\nMassaro" ) ;
    else
        printf ( "\nBennarivo" ) ;
}
```

(f)
```
main( )
{
    int  a = 40 ;
    if ( a > 40 && a < 45 )
        printf ( "a is greater than 40 and less than 45" ) ;
```

```
        else
            printf ( "%d", a ) ;
    }
```

(g)
```
    main( )
    {
        int  p = 8, q = 20 ;
        if ( p == 5 && q > 5 )
            printf ( "\nWhy not C" ) ;
        else
            printf ( "\nDefinitely C !" ) ;
    }
```

(h)
```
    main( )
    {
        int i = -1, j = 1, k ,l ;
        k = i && j ;
        l = i || j ;
        printf ( "%d %d", I, j ) ;
    }
```

(i)
```
    main( )
    {
        int x = 20 , y = 40 , z = 45 ;
        if ( x > y && x > z )
            printf( "x is big" ) ;
        else if ( y > x && y > z )
            printf( "y is big" ) ;
        else if ( z > x && z > y )
            printf( "z is big" ) ;
    }
```

(j)
```
    main( )
    {
        int i = -1, j = 1, k ,l ;
        k = !i && j ;
        l = !i  || j ;
        printf ( "%d %d", i, j ) ;
```

```
    }
```

(k)
```
main( )
{
    int  j = 4, k ;
    k = !5 && j ;
    printf ( "\nk = %d", k ) ;
}
```

[E] Point out the errors, if any, in the following programs:

(a)
```
/* This program
/* is an example of
/* using Logical operators */
main( )
{
    int  i = 2, j = 5 ;
    if ( i == 2 && j == 5 )
        printf ( "\nSatisfied at last" ) ;
}
```

(b)
```
main( )
{
    int  code, flag ;
    if ( code == 1 & flag == 0 )
        printf ( "\nThe eagle has landed" ) ;
}
```

(c)
```
main( )
{
    char  spy = 'a', password = 'z' ;
    if ( spy == 'a' or password == 'z' )
        printf ( "\nAll the birds are safe in the nest" ) ;
}
```

(d)
```
main( )
{
```

```
        int i = 10, j = 20 ;
        if ( i = 5 ) && if ( j = 10 )
            printf ( "\nHave a nice day" ) ;
    }
```

(a)
```
    main( )
    {
        int x = 10 , y = 20;
        if ( x >= 2 and y <=50 )
            printf ( "\n%d", x ) ;
    }
```

(b)
```
    main( )
    {
        int a, b ;
        if ( a == 1 & b == 0 )
            printf ( "\nGod is Great" ) ;
    }
```

(c)
```
    main( )
    {
        int x = 2;
        if ( x == 2 && x != 0 ) ;
        {
            printf ( "\nHi" ) ;
            printf( "\nHello" ) ;
        }
        else
            printf( "Bye" ) ;
    }
```

(d)
```
    main( )
    {
        int i = 10, j = 10 ;
        if ( i && j == 10)
            printf ( "\nHave a nice day" ) ;
    }
```

[F] Attempt the following:

(a) Any year is entered through the keyboard, write a program to determine whether the year is leap or not. Use the logical operators **&&** and **||**.

(b) Any character is entered through the keyboard, write a program to determine whether the character entered is a capital letter, a small case letter, a digit or a special symbol.

The following table shows the range of ASCII values for various characters.

Characters	**ASCII Values**
A – Z	65 – 90
a – z	97 – 122
0 – 9	48 – 57
special symbols	0 - 47, 58 - 64, 91 - 96, 123 - 127

(c) An Insurance company follows following rules to calculate premium.

(1) If a person's health is excellent and the person is between 25 and 35 years of age and lives in a city and is a male then the premium is Rs. 4 per thousand and his policy amount cannot exceed Rs. 2 lakhs.

(2) If a person satisfies all the above conditions except that the sex is female then the premium is Rs. 3 per thousand and her policy amount cannot exceed Rs. 1 lakh.

(3) If a person's health is poor and the person is between 25 and 35 years of age and lives in a village and is a male

then the premium is Rs. 6 per thousand and his policy cannot exceed Rs. 10,000.

(4) In all other cases the person is not insured.

Write a program to output whether the person should be insured or not, his/her premium rate and maximum amount for which he/she can be insured.

(d) A certain grade of steel is graded according to the following conditions:

(i) Hardness must be greater than 50
(ii) Carbon content must be less than 0.7
(iii) Tensile strength must be greater than 5600

The grades are as follows:

Grade is 10 if all three conditions are met
Grade is 9 if conditions (i) and (ii) are met
Grade is 8 if conditions (ii) and (iii) are met
Grade is 7 if conditions (i) and (iii) are met
Grade is 6 if only one condition is met
Grade is 5 if none of the conditions are met

Write a program, which will require the user to give values of hardness, carbon content and tensile strength of the steel under consideration and output the grade of the steel.

(e) A library charges a fine for every book returned late. For first 5 days the fine is 50 paise, for 6-10 days fine is one rupee and above 10 days fine is 5 rupees. If you return the book after 30 days your membership will be cancelled. Write a program to accept the number of days the member is late to return the book and display the fine or the appropriate message.

(f) If the three sides of a triangle are entered through the keyboard, write a program to check whether the triangle is valid or not. The triangle is valid if the sum of two sides is greater than the largest of the three sides.

(g) If the three sides of a triangle are entered through the keyboard, write a program to check whether the triangle is isosceles, equilateral, scalene or right angled triangle.

(h) In a company, worker efficiency is determined on the basis of the time required for a worker to complete a particular job. If the time taken by the worker is between 2 – 3 hours, then the worker is said to be highly efficient. If the time required by the worker is between 3 – 4 hours, then the worker is ordered to improve speed. If the time taken is between 4 – 5 hours, the worker is given training to improve his speed, and if the time taken by the worker is more than 5 hours, then the worker has to leave the company. If the time taken by the worker is input through the keyboard, find the efficiency of the worker.

(i) A university has the following rules for a student to qualify for a degree with A as the main subject and B as the subsidiary subject:
   (a) He should get 55 percent or more in A and 45 percent or more in B.
   (b) If he gets than 55 percent in A he should get 55 percent or more in B. However, he should get at least 45 percent in A.
   (c) If he gets less than 45 percent in B and 65 percent or more in A he is allowed to reappear in an examination in B to qualify.
   (d) In all other cases he is declared to have failed.

   Write a program to receive marks in A and B and Output whether the student has passed, failed or is allowed to reappear in B.

(j) The policy followed by a company to process customer orders is given by the following rules:

(a) If a customer order is less than or equal to that in stock and has credit is OK, supply has requirement.

(b) If has credit is not OK do not supply. Send him intimation.

(c) If has credit is Ok but the item in stock is less than has order, supply what is in stock. Intimate to him data the balance will be shipped.

Write a C program to implement the company policy.

Conditional operators

[G] What would be the output of the following programs:

(a)
```
main( )
{
    int  i = -4, j, num ;
    j = ( num < 0 ? 0 : num * num ) ;
    printf ( "\n%d", j ) ;
}
```

(b)
```
main( )
{
    int  k, num = 30 ;
    k = ( num > 5 ? ( num <= 10 ? 100 : 200 ) : 500 ) ;
    printf ( "\n%d", num ) ;
}
```

(c)
```
main( )
{
    int  j = 4 ;
    ( !j != 1 ? printf ( "\nWelcome") : printf ( "\nGood Bye") ) ;
```

```
    }
```

[H] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    int  tag = 0, code = 1 ;
    if ( tag == 0 )
        ( code > 1 ? printf ( "\nHello" ) ? printf ( "\nHi" ) ) ;
    else
        printf ( "\nHello Hi !!" ) ;
}
```

(b)
```
main( )
{
    int  ji = 65 ;
    printf ( "\nji >= 65 ? %d : %c", ji ) ;
}
```

(c)
```
main( )
{
    int  i = 10, j ;
    i >= 5 ? ( j = 10 ) : ( j = 15 ) ;
    printf ( "\n%d %d", i, j ) ;
}
```

(d)
```
main( )
{
    int a = 5 , b = 6 ;
    ( a == b ? printf( "%d",a) ) ;
}
```

(e)
```
main( )
{
    int n = 9 ;
    ( n == 9 ? printf( "You are correct" ) ; : printf( "You are wrong" ) ;) ;
}
```

(f)
```
main( )
{
    int  kk = 65 ,ll ;
    ll = ( kk == 65 : printf ( "\n kk is equal to 65" ) : printf ( "\n kk is not
equal to 65" ) ) ;
    printf( "%d", ll ) ;
}
```

(g)
```
main( )
{
    int  x = 10, y = 20 ;
    x == 20 && y != 10 ? printf( "True" ) : printf( "False" ) ;
}
```

[I] Rewrite the following programs using conditional operators.

(a)
```
main( )
{
    int  x, min, max ;
    scanf ( "\n%d %d", &max, &x ) ;
    if ( x > max )
        max = x ;
    else
        min = x ;
}
```

(b)
```
main( )
{
    int  code ;
    scanf ( "%d", &code ) ;
    if ( code > 1 )
        printf ( "\nJerusalem" ) ;
    else
        if ( code < 1 )
            printf ( "\nEddie" ) ;
        else
            printf ( "\nC Brain" ) ;
```

(c)
```
main( )
{
    float sal ;
    printf ("Enter the salary" ) ;
    scanf ( "%f", &sal ) ;
    if ( sal < 40000 && sal > 25000 )
        printf ( "Manager" ) ;
    else
        if ( sal < 25000 && sal > 15000 )
            printf ( "Accountant" ) ;
        else
            printf ( "Clerk" ) ;
}
```

[J] Attempt the following:

(a) Using conditional operators determine:

(1) Whether the character entered through the keyboard is a lower case alphabet or not.

(2) Whether a character entered through the keyboard is a special symbol or not.

(b) Write a program using conditional operators to determine whether a year entered through the keyboard is a leap year or not.

(c) Write a program to find the greatest of the three numbers entered through the keyboard using conditional operators.

3 The Loop Control Structure

- Loops
- The *while* Loop
  - Tips and Traps
  - More Operators
- The *for* Loop
  - Nesting of Loops
  - Multiple Initialisations in the *for* Loop
- The Odd Loop
- The *break* Statement
  - The *continue* Statement
- The *do-while* Loop
- Summary
- Exercise

The programs that we have developed so far used either a sequential or a decision control instruction. In the first one, the calculations were carried out in a fixed order, while in the second, an appropriate set of instructions were executed depending upon the outcome of the condition being tested (or a logical decision being taken).

These programs were of limited nature, because when executed, they always performed the same series of actions, in the same way, exactly once. Almost always, if something is worth doing, it's worth doing more than once. You can probably think of several examples of this from real life, such as eating a good dinner or going for a movie. Programming is the same; we frequently need to perform an action over and over, often with variations in the details each time. The mechanism, which meets this need, is the 'loop', and loops are the subject of this chapter.

Loops

The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. This repetitive operation is done through a loop control instruction.

There are three methods by way of which we can repeat a part of a program. They are:

(a) Using a **for** statement
(b) Using a **while** statement
(c) Using a **do-while** statement

Each of these methods is discussed in the following pages.

The *while* Loop

It is often the case in programming that you want to do something a fixed number of times. Perhaps you want to calculate gross salaries of ten different persons, or you want to convert temperatures from centigrade to fahrenheit for 15 different cities. The **while** loop is ideally suited for such cases. Let us look at a simple example, which uses a **while** loop. The flowchart shown below would help you to understand the operation of the **while** loop.

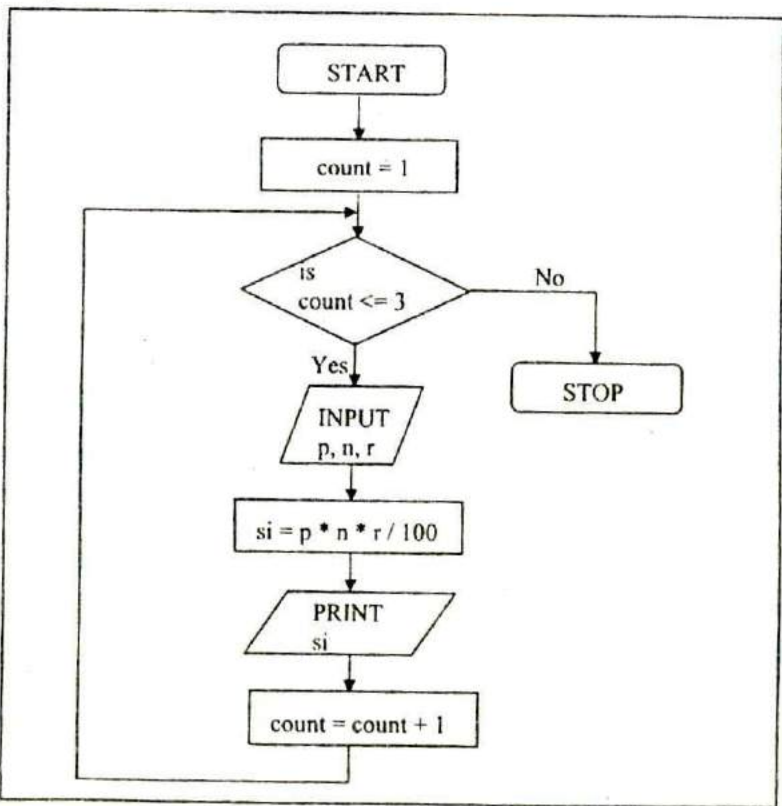


Figure 3.1

```
/* Calculation of simple interest for 3 sets of p, n and r */
main( )
{
    int  p, n, count ;
    float  r, si ;

    count = 1 ;
    while ( count <= 3 )
    {
        printf ( "\nEnter values of p, n and r " ) ;
        scanf ( "%d %d %f", &p, &n, &r ) ;
        si = p * n * r / 100 ;
        printf ( "Simple interest = Rs. %f", si ) ;

        count = count + 1 ;
    }
}
```

And here are a few sample runs...

```
Enter values of p, n and r  1000  5  13.5
Simple interest = Rs. 675.000000
Enter values of p, n and r  2000  5  13.5
Simple interest = Rs. 1350.000000
Enter values of p, n and r  3500  5  3.5
Simple interest = Rs. 612.500000
```

The program executes all statements after the **while** 3 times. The logic for calculating the simple interest is written within a pair of braces immediately after the **while** keyword. These statements form what is called the 'body' of the **while** loop. The parentheses after the **while** contain a condition. So long as this condition remains true all statements within the body of the **while** loop keep getting executed repeatedly. To begin with the variable **count** is initialized to 1 and every time the simple interest logic is executed the value of **count** is incremented by one. The variable **count** is many a times called either a 'loop counter' or an 'index variable'.

The operation of the **while** loop is illustrated in the following figure.

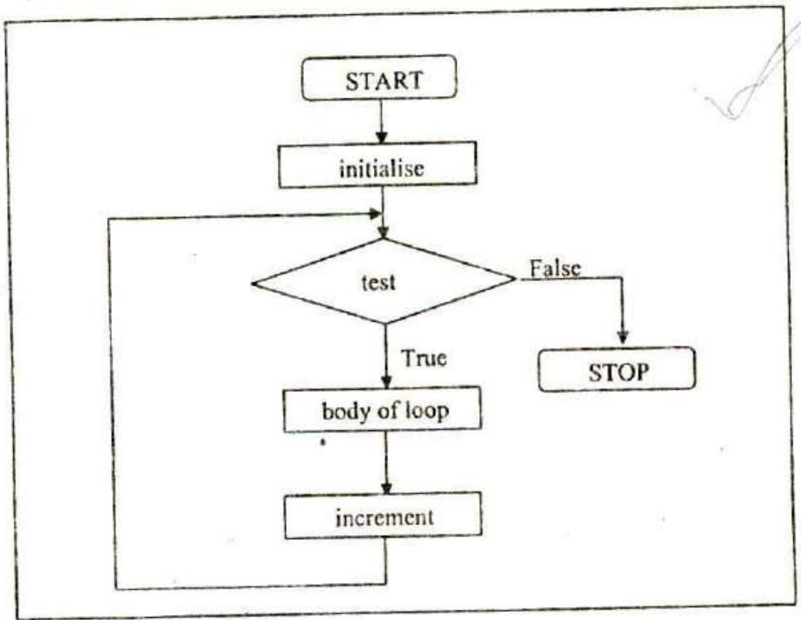


Figure 3.2

Tips and Traps

The general form of **while** is as shown below:

```
initialise loop counter ;
while ( test loop counter using a condition )
{
    do this ;
    and this ;
    increment loop counter ;
}
```

Note the following points about **while...**

- The statements within the **while** loop would keep on getting executed till the condition being tested remains true. When the condition becomes false, the control passes to the first statement that follows the body of the **while** loop.

  In place of the condition there can be any other valid expression. So long as the expression evaluates to a non-zero value the statements within the loop would get executed.

- The condition being tested may use relational or logical operators as shown in the following examples:

```
while ( i <= 10 )
while ( i >= 10 && j <= 15 )
while ( j > 10 && ( b < 15 || c < 20 ) )
```

- The statements within the loop may be a single line or a block of statements. In the first case the parentheses are optional. For example,

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

  is same as

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

- As a rule the while must test a condition that will eventually become false, otherwise the loop would be executed forever, indefinitely.

```
main( )
{
    int  i = 1 ;
```

```
    while ( i <= 10 )
        printf ( "%d\n", i ) ;
}
```

This is an indefinite loop, since **i** remains equal to 1 forever. The correct form would be as under:

```
main( )
{
    int  i = 1 ;
    while ( i <= 10 )
    {
        printf ( "%d\n", i ) ;
        i = i + 1 ;
    }
}
```

– Instead of incrementing a loop counter, we can even decrement it and still manage to get the body of the loop executed repeatedly. This is shown below:

```
main( )
{
    int  i = 5 ;
    while ( i >= 1 )
    {
        printf ( "\nMake the computer literate!" ) ;
        i = i - 1 ;
    }
}
```

– It is not necessary that a loop counter must only be an **int**. It can even be a **float**.

```
main( )
{
    float  a = 10.0 ;
    while ( a <= 10.5 )
    {
```

```
        printf ( "\nRaindrops on roses..." )
        printf ( "...and whiskers on kittens" ) ;
        a = a + 0.1 ;
    }
}
```

- Even floating point loop counters can be decremented. Once again the increment and decrement could be by any value, not necessarily 1.

  What do you think would be the output of the following program?

```
main( )
{
    int  i = 1 ;
    while ( i <= 32767 )
    {
        printf ( "%d\n", i ) ;
        i = i + 1 ;
    }
}
```

  No, it doesn't print numbers from 1 to 32767. It's an indefinite loop. To begin with, it prints out numbers from 1 to 32767. After that value of **i** is incremented by 1, therefore it tries to become 32768, which falls outside the valid integer range, so it goes to other side and becomes -32768 which would certainly satisfy the condition in the **while**. This process goes on indefinitely.

- What will be the output of the following program?

```
main( )
{
    int  i = 1 ;
    while ( i <= 10 ) ;
    {
        printf ( "%d\n", i ) ;
```

```
        i = i + 1 ;
    }
}
```

This is another indefinite loop, and it doesn't give any output at all. The reason is, we have carelessly given a ; after the **while**. This would make the loop work like this...

```
while ( i <= 10 )
    ;
{
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

Since the value of **i** is not getting incremented the control would keep rotating within the loop, eternally. Note that enclosing **printf( )** and **i = i +1** within a pair of braces is not an error. In fact we can put a pair of braces around any individual statement or set of statements without affecting the execution of the program.

More Operators

There are variety of operators which are frequently used with **while**. To illustrate their usage let us consider a problem wherein numbers from 1 to 10 are to be printed on the screen. The program for performing this task can be written using **while** in the following different ways:

```
(a) main( )
    {
        int  i = 1 ;
        while ( i <= 10 )
        {
            printf ( "%d\n", i ) ;
            i = i + 1 ;
```

```
    }
}
```

(b)
```
main( )
{
    int i = 1 ;
    while ( i <= 10 )
    {
        printf ( "%d\n", i ) ;
        i++ ;
    }
}
```

Note that the increment operator ++ increments the value of **i** by 1, every time the statement **i++** gets executed. Similarly, to reduce the value of a variable by 1 a decrement operator -- is also available.

However, never use **n+++** to increment the value of **n** by 2, since C doesn't recognize the operator +++.

(c)
```
main( )
{
    int i = 1 ;
    while ( i <= 10 )
    {
        printf ( "%d\n", i ) ;
        i += 1 ;
    }
}
```

Note that += is a compound assignment operator. It increments the value of **i** by 1. Similarly, **j = j + 10** can also be written as **j += 10**. Other compound assignment operators are -=, *=, / = and %=.

(d)
```
main( )
```

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

In the statement **while (i++ < 10)**, firstly the comparison of value of **i** with 10 is performed, and then the incrementation of **i** takes place. Since the incrementation of **i** happens after its usage, here the ++ operator is called a post-incrementation operator. When the control reaches **printf( )**, **i** has already been incremented, hence **i** must be initialized to 0.

(e)
```
main( )
{
    int i = 0 ;
    while ( ++i <= 10 )
        printf ( "%d\n", i ) ;
}
```

In the statement **while (++i <= 10)**, firstly incrementation of **i** takes place, then the comparison of value of **i** with 10 is performed. Since the incrementation of **i** happens before its usage, here the ++ operator is called a pre-incrementation operator.

The *for* Loop

Perhaps one reason why few programmers use **while** is that they are too busy using the **for**, which is probably the most popular looping instruction. The **for** allows us to specify three things about a loop in a single line:

(a) Setting a loop counter to an initial value.
(b) Testing the loop counter to determine whether its value has reached the number of repetitions desired.

(c) Increasing the value of loop counter each time the program segment within the loop has been executed.

The general form of **for** statement is as under:

```
for ( initialise counter ; test counter ; increment counter )
{
    do this ;
    and this ;
    and this ;
}
```

Let us write down the simple interest program using **for**. Compare this program with the one, which we wrote using **while**. The flowchart is also given below for a better understanding.

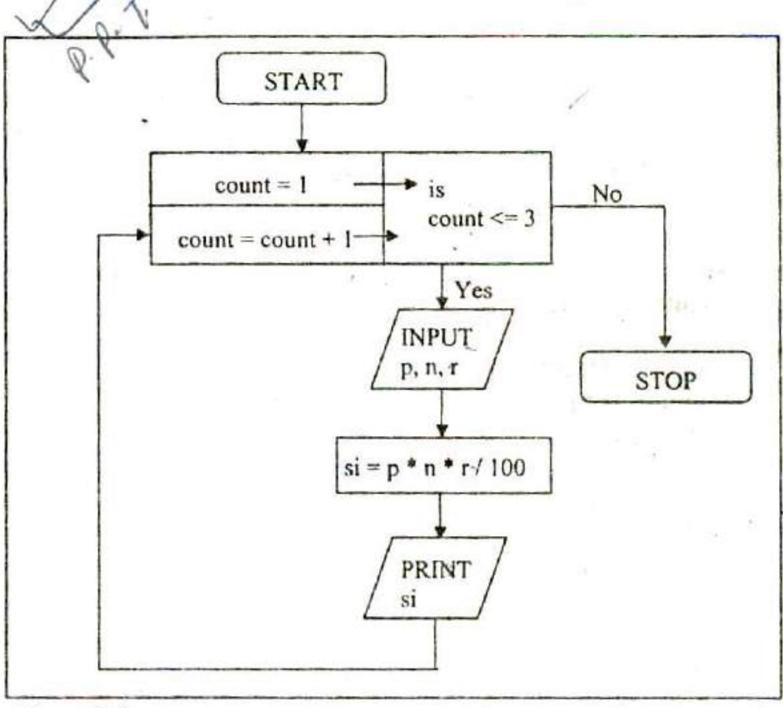


Figure 3.3

```
/* Calculation of simple interest for 3 sets of p, n and r */
main ( )
{
    int  p, n, count ;
    float  r, si ;

    for ( count = 1 ; count <= 3 ; count = count + 1 )
    {
        printf ( "Enter values of p, n, and r " ) ;
        scanf ( "%d %d %f", &p, &n, &r ) ;

        si = p * n * r / 100 ;
        printf ( "Simple Interest = Rs.%f\n", si ) ;
    }
```

If this program is compared with the one written using **while**, it can be seen that the three steps—initialization, testing and incrementation—required for the loop construct have now been incorporated in the **for** statement.

Let us now examine how the **for** statement gets executed:

- When the **for** statement is executed for the first time, the value of **count** is set to an initial value 1.

- Now the condition **count** <= 3 is tested. Since **count** is 1 the condition is satisfied and the body of the loop is executed for the first time.

- Upon reaching the closing brace of **for**, control is sent back to the **for** statement, where the value of **count** gets incremented by 1.

- Again the test is performed to check whether the new value of **count** exceeds 3.

- If the value of **count** is still within the range 1 to 3, the statements within the braces of **for** are executed again.

- The body of the **for** loop continues to get executed till **count** doesn't exceed the final value 3.

- When **count** reaches the value 4 the control exits from the loop and is transferred to the statement (if any) immediately after the body of **for**.

The following figure would help in further clarifying the concept of execution of the **for** loop.

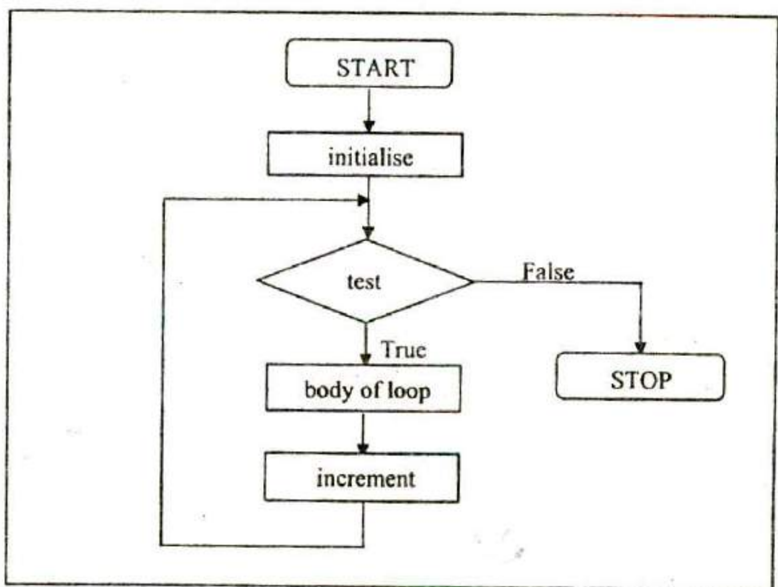


Figure 3.4

It is important to note that the initialization, testing and incrementation part of a **for** loop can be replaced by any valid expression. Thus the following **for** loops are perfectly ok.

```
for ( i = 10 ; i ; i -- )
    printf ( "%d", i ) ;
for ( i < 4 ; j = 5 ; j = 0 )
    printf ( "%d", i ) ;
for ( i = 1; i <=10 ; printf ( "%d",i++ )
    ;
for ( scanf ( "%d", &i ) ; i <= 10 ; i++ )
    printf ( "%d", i ) ;
```

Let us now write down the program to print numbers from 1 to 10 in different ways. This time we would use a **for** loop instead of a **while** loop.

(a)
```
main( )
{
    int  i ;
    for ( i = 1 ; i <= 10 ; i = i + 1 )
        printf ( "%d\n", i ) ;
}
```

Note that the initialisation, testing and incrementation of loop counter is done in the **for** statement itself. Instead of **i = i + 1**, the statements **i++** or **i += 1** can also be used.

Since there is only one statement in the body of the **for** loop, the pair of braces have been dropped. As with the **while**, the default scope of **for** is the immediately next statement after **for**.

(b)
```
main( )
{
    int  i ;
    for ( i = 1 ; i <= 10 ; )
    {
        printf ( "%d\n", i ) ;
        i = i + 1 ;
    }
}
```

Here, the incrementation is done within the body of the **for** loop and not in the **for** statement. Note that inspite of this the semicolon after the condition is necessary.

(c)
```
main( )
{
    int  i = 1 ;
    for ( ; i <= 10 ; i = i + 1 )
        printf ( "%d\n", i ) ;
}
```

Here the initialisation is done in the declaration statement itself, but still the semicolon before the condition is necessary.

(d)
```
main( )
{
    int  i = 1 ;
    for ( ; i <= 10 ; )
    {
        printf ( "%d\n", i ) ;
        i = i + 1 ;
    }
}
```

Here, neither the initialisation, nor the incrementation is done in the **for** statement, but still the two semicolons are necessary.

(e)
```
main( )
{
    int  i ;
    for ( i = 0 ; i++ < 10 ; )
        printf ( "%d\n", i ) ;
}
```

Here, the comparison as well as the incrementation is done through the same statement, **i++ < 10**. Since the **++** operator comes after **i** firstly comparison is done, followed by incrementation. Note that it is necessary to initialize **i** to 0.

(f)
```
main( )
{
    int  i ;
    for ( i = 0 ; ++i <= 10 ; )
        printf ( "%d\n", i ) ;
}
```

Here, both, the comparison and the incrementation is done through the same statement, **++i <= 10**. Since ++ precedes **i** firstly incrementation is done, followed by comparison. Note that it is necessary to initialize **i** to 0.

Nesting of Loops

The way **if** statements can be nested, similarly **whiles** and **fors** can also be nested. To understand how nested loops work, look at the program given below:

```
/* Demonstration of nested loops */
main( )
{
    int  r, c, sum ;
    for ( r = 1 ; r <= 3 ; r++ ) /* outer loop */
    {
        for ( c = 1 ; c <= 2 ; c++ ) /* inner loop */
        {
            sum = r + c ;
            printf ( "r = %d c = %d sum = %d\n", r, c, sum ) ;
        }
    }
}
```

When you run this program you will get the following output:

```
r = 1 c = 1 sum = 2
r = 1 c = 2 sum = 3
r = 2 c = 1 sum = 3
r = 2 c = 2 sum = 4
r = 3 c = 1 sum = 4
r = 3 c = 2 sum = 5
```

Here, for each value of **r** the inner loop is cycled through twice, with the variable **c** taking values from 1 to 2. The inner loop

terminates when the value of c exceeds 2, and the outer loop terminates when the value of r exceeds 3.

As you can see, the body of the outer **for** loop is indented, and the body of the inner **for** loop is further indented. These multiple indentations make the program easier to understand.

Instead of using two statements, one to calculate **sum** and another to print it out, we can compact this into one single statement by saying:

```
printf ( "r = %d c = %d sum = %d\n", r, c, r + c ) ;
```

The way **for** loops have been nested here, similarly, two **while** loops can also be nested. Not only this, a **for** loop can occur within a **while** loop, or a **while** within a **for**.

Multiple Initialisations in the *for* Loop

The initialisation expression of the **for** loop can contain more than one statement separated by a comma. For example,

```
for ( i = 1, j = 2 ; j <= 10 ; j++ )
```

Multiple statements can also be used in the incrementation expression of **for** loop; i.e., you can increment (or decrement) two or more variables at the same time. However, only one expression is allowed in the test expression. This expression may contain several conditions linked together using logical operators.

Use of multiple statements in the initialisation expression also demonstrates why semicolons are used to separate the three expressions in the **for** loop. If commas had been used, they could not also have been used to separate multiple statements in the initialisation expression, without confusing the compiler.

The Odd Loop

The loops that we have used so far executed the statements within them a finite number of times. However, in real life programming one comes across a situation when it is not known beforehand how many times the statements in the loop are to be executed. This situation can be programmed as shown below:

```
/* Execution of a loop an unknown number of times */
main( )
{
    char another ;
    int num ;
    do
    {
        printf ( "Enter a number " ) ;
        scanf ( "%d", &num ) ;
        printf ( "square of %d is %d", num, num * num ) ;
        printf ( "\nWant to enter another number y/n " ) ;
        scanf ( " %c", &another ) ;
    } while ( another == 'y' ) ;
}
```

And here is the sample output...

```
Enter a number 5
square of 5 is 25
Want to enter another number y/n y
Enter a number 7
square of 7 is 49
Want to enter another number y/n n
```

In this program the **do-while** loop would keep getting executed till the user continues to answer y. The moment he answers n, the loop terminates, since the condition (**another == 'y'**) fails. Note that this loop ensures that statements within it are executed at least once even if **n** is supplied first time itself.

Though it is simpler to program such a requirement using a **do-while** loop, the same functionality if required, can also be accomplished using **for** and **while** loops as shown below:

```
/* odd loop using a for loop */
main( )
{
    char another = 'y' ;
    int num ;
    for ( ; another == 'y' ; )
    {
        printf ( "Enter a number " ) ;
        scanf ( "%d", &num ) ;
        printf ( "square of %d is %d", num, num * num ) ;
        printf ( "\nWant to enter another number y/n " ) ;
        scanf ( " %c", &another ) ;
    }
}
```

```
/* odd loop using a while loop */
main( )
{
    char another = 'y' ;
    int num ;

    while ( another == 'y' )
    {
        printf ( "Enter a number " ) ;
        scanf ( "%d", &num ) ;
        printf ( "square of %d is %d", num, num * num ) ;
        printf ( "\nWant to enter another number y/n " ) ;
        scanf ( " %c", &another ) ;
    }
}
```

The *break* Statement

We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test. The keyword **break** allows us to do this. When **break** is encountered inside any loop, control automatically passes to the first statement after the loop. A **break** is usually associated with an **if**. As an example, let's consider the following example.

Example: Write a program to determine whether a number is prime or not. A prime number is one, which is divisible only by 1 or itself.

All we have to do to test whether a number is prime or not, is to divide it successively by all numbers from 2 to one less than itself. If remainder of any of these divisions is zero, the number is not a prime. If no division yields a zero then the number is a prime number. Following program implements this logic.

```
main( )
{
    int  num, i ;

    printf ( "Enter a number " ) ;
    scanf ( "%d", &num ) ;

    i = 2 ;
    while ( i <= num - 1 )
    {
        if ( num % i == 0 )
        {
            printf ( "Not a prime number" ) ;
            break ;
        }
        i++ ;
    }
```

```
    if ( i == num )
        printf ( "Prime number" ) ;
}
```

In this program the moment **num % i** turns out to be zero, (i.e. **num** is exactly divisible by **i**) the message "Not a prime number" is printed and the control breaks out of the **while** loop. Why does the program require the **if** statement after the **while** loop at all? Well, there are two ways the control could have reached outside the **while** loop:

(a) It jumped out because the number proved to be not a prime.
(b) The loop came to an end because the value of **i** became equal to **num**.

When the loop terminates in the second case, it means that there was no number between 2 to **num - 1** that could exactly divide **num**. That is, **num** is indeed a prime. If this is true, the program should print out the message "Prime number".

The keyword **break**, breaks the control only from the **while** in which it is placed. Consider the following program, which illustrates this fact.

```
main( )
{
    int i = 1 , j = 1 ;

    while ( i++ <= 100 )
    {
        while ( j++ <= 200 )
        {
            if ( j == 150 )
                break ;
            else
                printf ( "%d %d\n", i, j ) ;
        }
```

```
    }
}
```

In this program when **j** equals 150, **break** takes the control outside the inner **while** only, since it is placed inside the inner **while**.

The *continue* Statement

In some programming situations we want to take the control to the beginning of the loop, bypassing the statements inside the loop, which have not yet been executed. The keyword **continue** allows us to do this. When **continue** is encountered inside any loop, control automatically passes to the beginning of the loop.

A **continue** is usually associated with an **if**. As an example, let's consider the following program.

```
main( )
{
    int i, j ;

    for ( i = 1 ; i <= 2 ; i++ )
    {
        for ( j = 1 ; j <= 2 ; j++ )
        {
            if ( i == j )
                continue ;

            printf ( "\n%d %d\n", i, j ) ;
        }
    }
}
```

The output of the above program would be...

```
1 2
2 1
```

Note that when the value of **i** equals that of **j**, the **continue** statement takes the control to the **for** loop (inner) bypassing rest of the statements pending execution in the **for** loop (inner).

The *do-while* Loop

The **do-while** loop looks like this:

```
do
{
    this ;
    and this ;
    and this ;
    and this ;
} while ( this condition is true ) ;
```

There is a minor difference between the working of **while** and **do-while** loops. This difference is the place where the condition is tested. The **while** tests the condition before executing any of the statements within the **while** loop. As against this, the **do-while** tests the condition after having executed the statements within the loop. Figure 3.5 would clarify the execution of **do-while** loop still further.

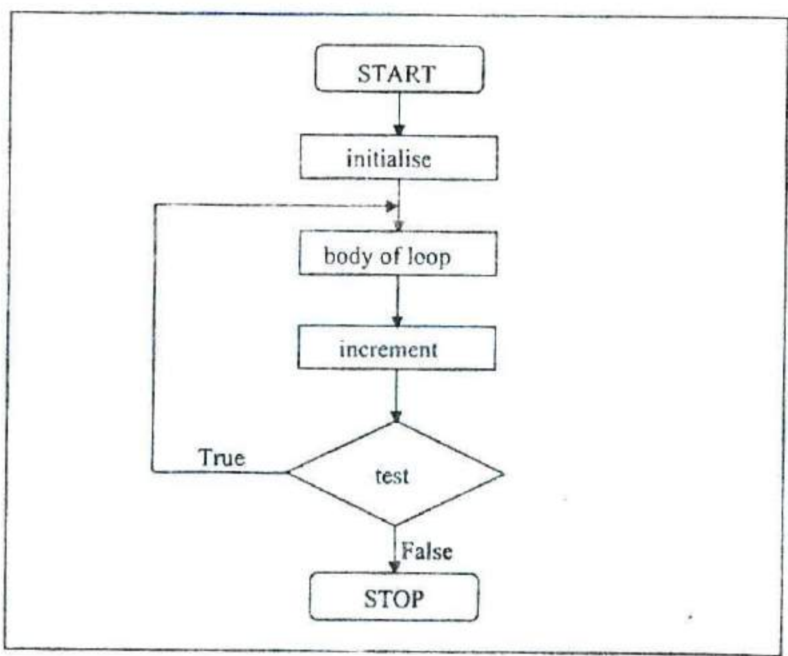


Figure 3.5

This means that **do-while** would execute its statements at least once, even if the condition fails for the first time. The **while**, on the other hand will not execute its statements if the condition fails for the first time. This difference is brought about more clearly by the following program.

```
main( )
{
    while ( 4 < 1 )
        printf ( "Hello there \n") ;
}
```

Here, since the condition fails the first time itself, the **printf( )** will not get executed at all. Let's now write the same program using a **do-while** loop.

```
main( )
{
    do
    {
        printf ( "Hello there \n") ;
    } while ( 4 < 1 ) ;
}
```

In this program the **printf( )** would be executed once, since first the body of the loop is executed and then the condition is tested.

There are some occasions when we want to execute a loop at least once no matter what. This is illustrated in the following example:

break and **continue** are used with **do-while** just as they would be in a **while** or a **for** loop. A **break** takes you out of the **do-while** bypassing the conditional test. A **continue** sends you straight to the test at the end of the loop.

Summary

(a) The three type of loops available in C are **for**, **while**, and **do-while**.
(b) A **break** statement takes the execution control out of the loop.
(c) A **continue** statement skips the execution of the statements after it and takes the control to the beginning of the loop.
(d) A **do-while** loop is used to ensure that the statements within the loop are executed at least once.
(e) The ++ operator increments the operand by 1, whereas, the -- operator decrements it by 1.
(f) The operators +=, -=, *=, /=, %= are compound assignment operators. They modify the value of the operand to the left of them.

Exercise

while Loop

[A] What would be the output of the following programs:

(a)
```
main( )
{
    int j ;
    while ( j <= 10 )
    {
        printf ( "\n%d", j ) ;
        j = j + 1 ;
    }
}
```

(b)
```
main( )
{
    int i = 1 ;
    while ( i <= 10 ) ;
    {
        printf ( "\n%d", i ) ;
```

1 - 10

```
        i++ ;
    }
}
```

(c)
```
main( )
{
    int j ;
    while ( j <= 10 )
    {
        printf ( "\n%d", j ) ;
        j = j + 1 ;
    }
}
```

(d)
```
main( )
{
    int x = 1 ;
    while ( x == 1 )
    {
        x = x - 1 ;
        printf ( "\n%d", x ) ;
    }
}
```

(e)
```
main( )
{
    int x = 1 ;
    while ( x == 1 )
        x = x - 1 ;
    printf ( "\n%d", x ) ;
}
```

(f)
```
main( )
{
    char x ;
```

```
        while ( x = 0 ; x <= 255 ; x++ )
            printf ( "\nAscii value %d Character %c", x, x ) ;
    }
```

(g)
```
    main( )
    {
        int  x = 4, y, z ;
        y = --x ;
        z = x-- ;
        printf ( "\n%d %d %d", x, y, z ) ;
    }
```

(h)
```
    main( )
    {
        int  x = 4, y = 3, z ;

        z = x-- -y ;
        printf ( "\n%d %d %d", x, y, z ) ;
    }
```

(i)
```
    main( )
    {
        while ( 'a' < 'b' )
            printf ( "\nmalyalam is a palindrome" ) ;
    }
```

(j)
```
    main( )
    {
        int  i = 10 ;
        while ( i = 20 )
            printf ( "\nA computer buff!" ) ;
    }
```

(k)
```
    main( )
    {
        int  i ;
        while ( i = 10 )
        {
```

```
            printf ( "\n%d", i ) ;
            i = i + 1 ;
        }
    }
```

(l)
```
    main( )
    {
        float x = 1.1 ;
        while ( x == 1.1 )
        {
            printf ( "\n%f", x ) ;
            x = x – 0.1 ;
        }
    }
```

(m)
```
    main( )
    {
        while ( '1' < '2' )
            printf ( "\nIn while loop" ) ;
    }
```

(n)
```
    main( )
    {
        char x ;
        for ( x = 0 ; x <= 255 ; x++ )
            printf ( "\nAscii value %d Character %c", x, x ) ;
    }
```

(o)
```
    main( )
    {
        int x = 4, y = 0, z ;
        while ( x >= 0 )
        {
            x-- ;
            y++ ;
            if ( x == y )
```

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

(p)
```
main( )
{
    int x = 4, y = 0, z ;
    while ( x >= 0 )
    {
        if ( x == y )
            break ;
        else
            printf ( "\n%d %d", x, y ) ;
        x-- ;
        y++ ;
    }
}
```

[B] Attempt the following:

(a) Write a program to calculate overtime pay of 10 employees. Overtime is paid at the rate of Rs. 12.00 per hour for every hour worked above 40 hours. Assume that employees do not work for fractional part of an hour.

(b) Write a program to find the factorial value of any number entered through the keyboard.

(c) Two numbers are entered through the keyboard. Write a program to find the value of one number raised to the power of another.

(d) Write a program to print all the ASCII values and their equivalent characters using a **while** loop. The ASCII values vary from 0 to 255.

(e) Write a program to print out all Armstrong numbers between 1 and 500. If sum of cubes of each digit of the number is equal to the number itself, then the number is called an Armstrong number. For example, 153 = (1 * 1 * 1) + (5 * 5 * 5) + (3 * 3 * 3)

(f) Write a program for a matchstick game being played between the computer and a user. Your program should ensure that the computer always wins. Rules for the game are as follows:

- There are 21 matchsticks.
- The computer asks the player to pick 1, 2, 3, or 4 matchsticks.
- After the person picks, the computer does its picking.
- Whoever is forced to pick up the last matchstick loses the game.

(g) Write a program to enter the numbers till the user wants and at the end it should display the count of positive, negative and zeros entered.

(h) Write a program to find the octal equivalent of the entered number.

(i) Write a program to find the range of a set of numbers. Range is the difference between the smallest and biggest number in the list.

for, break, continue, do-while

[C] What would be the output of the following programs:

(a)
```
main( )
{
    int i = 0 ;
    for ( ; i ; )
```

```
        printf ( "\nHere is some mail for you" ) ;
}
```

(b)
```
main( )
{
    int i ;
    for ( i = 1 ; i <= 5 ; printf ( "\n%d", i ) ) ;
        i++ ;
}
```

(c)
```
main( )
{
    int i = 1, j = 1 ;
    for ( ; ; )
    {
        if ( i > 5 )
            break ;
        else
            j += i ;
        printf ( "\n%d", j ) ;
        i += j ;
    }
}
```

(d)
```
main( )
{
    int i ;
    for ( i = 1 ; i <= 5 ; printf ( "\n%c", 65 ) ) ;
        i++ ;
}
```

[D] Answer the following:

(a) The three parts of the loop expression in the **for** loop are:

the i____________ expression
the t____________ expression
the i____________ expression

(b) An expression contains relational operators, assignment operators, and arithmetic operators. In the absence of parentheses, they will be evaluated in which of the following order:

1. assignment, relational, arithmetic
2. arithmetic, relational, assignment
3. relational, arithmetic, assignment
4. assignment, arithmetic, relational

(c) The **break** statement is used to exit from:

1. an **if** statement
2. a **for** loop
3. a program
4. the **main( )** function

(d) A **do-while** loop is useful when we want that the statements within the loop must be executed:

1. Only once
2. At least once
3. More than once
4. None of the above

(e) In what sequence the initialization, testing and execution of body is done in a do-while loop

1. Initialization, execution of body, testing
2. Execution of body, initialization, testing
3. Initialization, testing, execution of body
4. None of the above

(f) Which of the following is not an infinite loop.

1.
```
int i = 1 ;
while ( 1 )
{
  i++ ;
}
```

2.
```
for ( ; ; ) ;
```

```
3. int True = 0, false ;
   while ( True )
   {
     False = 1 ;
   }
```

```
4. int y, x = 0 ;
   do
   {
     y = x ;
   } while ( x == 0 ) ;
```

(g) Which of the following statement is used to take the control to the beginning of the loop?

1. exit
2. break
3. continue
4. None of the above

[E] Attempt the following:

(a) Write a program to print all prime numbers from 1 to 300. (Hint: Use nested loops, **break** and **continue**)

(b) Write a program to fill the entire screen with a smiling face. The smiling face has an ASCII value 1.

(c) Write a program to add first seven terms of the following series using a **for** loop:

$$\frac{1}{1!} + \frac{2}{2!} + \frac{3}{3!} + \ldots\ldots$$

(d) Write a program to generate all combinations of 1, 2 and 3 using **for** loop.

(e) According to a study, the approximate level of intelligence of a person can be calculated using the following formula:

i = 2 + (y + 0.5 x)

Write a program, which will produce a table of values of **i**, **y** and **x**, where **y** varies from 1 to 6, and, for each value of **y**, **x** varies from 5.5 to 12.5 in steps of 0.5.

(f) Write a program to produce the following output:

```
A B C D E F G F E D C B A
A B C D E F   F E D C B A
A B C D E       E D C B A
A B C D           D C B A
A B C               C B A
A B                   B A
A                       A
```

(g) Write a program to fill the entire screen with diamond and heart alternatively. The ASCII value for heart is 3 and that of diamond is 4.

(h) Write a program to print the multiplication table of the number entered by the user. The table should get displayed in the following form.

29 * 1 = 29
29 * 2 = 58
...

(i) Write a program to produce the following output:

1

(j) Write a program to produce the following output:

```
            1
        1       1
    1       2       1
1       3       3       1
1       4       6       4       1
```

(k) A machine is purchased which will produce earning of Rs. 1000 per year while it lasts. The machine costs Rs. 6000 and will have a salvage of Rs. 2000 when it is condemned. If 12 percent per annum can be earned on alternate investments what would be the minimum life of the machine to make it a more attractive investment compared to alternative investment?

(l) When interest compounds **q** times per year at an annual rate of **r** % for **n** years, the principle **p** compounds to an amount **a** as per the following formula

$$a = p (1 + r / q)^{nq}$$

Wr:.e a program to read 10 sets of **p, r, n** & **q** and calculate the corresponding as.

(m) The natural logarithm can be approximated by the following series.

$$\frac{x-1}{x} + \frac{1}{2}\left(\frac{x-1}{x}\right)^2 + \frac{1}{2}\left(\frac{x-1}{x}\right)^3 + \frac{1}{2}\left(\frac{x-1}{x}\right)^4 + \ldots$$

If **x** is input through the keyboard, write a program to calculate the sum of first seven terms of this series.

4 The Case Control Structure

- Decisions Using *switch*
  - The Tips and Traps
- *switch* Versus *if-else* Ladder
- The *goto* Keyword
- Summary
- Exercise

In real life we are often faced with situations where we are required to make a choice between a number of alternatives rather than only one or two. For example, which school to join or which hotel to visit or still harder which girl to marry (you almost always end up making a wrong decision is a different matter altogether!). Serious C programming is same; the choice we are asked to make is more complicated than merely selecting between two alternatives. C provides a special control statement that allows us to handle such cases effectively; rather than using a series of **if** statements. This control instruction is in fact the topic of this chapter. Towards the end of the chapter we would also study a keyword called **goto**, and understand why we should avoid its usage in C programming.

Decisions Using *switch*

The control statement that allows us to make a decision from the number of choices is called a **switch**, or more correctly a **switch-case-default**, since these three keywords go together to make up the control statement. They most often appear as follows:

```
switch ( integer expression )
{
    case constant 1 :
        do this ;
    case constant 2 :
        do this ;
    case constant 3 :
        do this ;
    default :
        do this ;
}
```

The integer expression following the keyword **switch** is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an

integer. The keyword **case** is followed by an integer or a character constant. Each constant in each **case** must be different from all the others. The "do this" lines in the above form of **switch** represent any valid C statement.

What happens when we run a program containing a **switch**? First, the integer expression following the keyword **switch** is evaluated. The value it gives is then matched, one by one, against the constant values that follow the **case** statements. When a match is found, the program executes the statements following that **case**, and all subsequent **case** and **default** statements as well. If no match is found with any of the **case** statements, only the statements following the **default** are executed. A few examples will show how this control structure works.

Consider the following program:

```
main( )
{
    int  i = 2 ;

    switch ( i )
    {
        case 1 :
            printf ( "I am in case 1 \n" ) ;
        case 2 :
            printf ( "I am in case 2 \n" ) ;
        case 3 :
            printf ( "I am in case 3 \n" ) ;
        default :
            printf ( "I am in default \n" ) ;
    }
}
```

The output of this program would be:

```
I am in case 2
```

```
I am in case 3
I am in default
```

The output is definitely not what we expected! We didn't expect the second and third line in the above output. The program prints case 2 and 3 and the default case. Well, yes. We said the **switch** executes the case where a match is found and all the subsequent cases and the **default** as well.

If you want that only case 2 should get executed, it is upto you to get out of the **switch** then and there by using a **break** statement. The following example shows how this is done. Note that there is no need for a **break** statement after the **default**, since the control comes out of the **switch** anyway.

```
main( )
{
    int  i = 2 ;

    switch ( i )
    {
        case 1 :
            printf ( "I am in case 1 \n" ) ;
            break ;
        case 2 :
            printf ( "I am in case 2 \n" ) ;
            break ;
        case 3 :
            printf ( "I am in case 3 \n" ) ;
            break ;
        default :
            printf ( "I am in default \n" ) ;
    }
}
```

The output of this program would be:

```
I am in case 2
```

The operation of **switch** is shown below in the form of a flowchart for a better understanding.

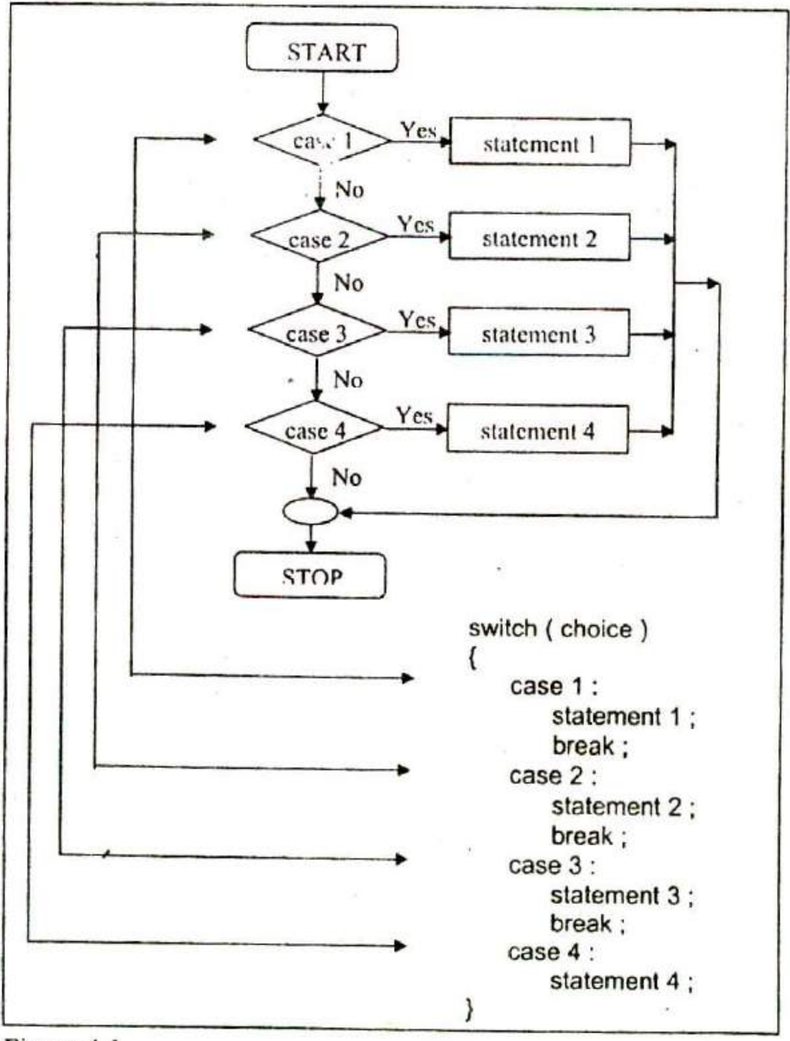


Figure 4.1

The Tips and Traps

A few useful tips about the usage of **switch** and a few pitfalls to be avoided:

(a) The earlier program that used **switch** may give you the wrong impression that you can use only cases arranged in ascending order, 1, 2, 3 and default. You can in fact put the cases in any order you please. Here is an example of scrambled case order:

```
main( )
{
   int i = 22 ;

   switch ( i )
   {
       case 121 :
           printf ( "I am in case 121 \n" ) ;
           break ;
       case 7 :
           printf ( "I am in case 7 \n" ) ;
           break ;
       case 22 :
           printf ( "I am in case 22 \n" ) ;
           break ;
       default :
           printf ( "I am in default \n" ) ;
   }
}
```

The output of this program would be:

```
I am in case 22
```

(b) You are also allowed to use **char** values in **case** and **switch** as shown in the following program:

```
main( )
```

```
{
    char  c = 'x' ;

    switch ( c )
    {
        case 'v' :
            printf ( "I am in case v \n" ) ;
            break ;
        case 'a' :
            printf ( "I am in case a \n" ) ;
            break ;
        case 'x' :
            printf ( "I am in case x \n" ) ;
            break ;
        default :
            printf ( "I am in default \n" ) ;
    }
}
```

The output of this program would be:

```
I am in case x
```

In fact here when we use 'v', 'a', 'x' they are actually replaced by the ASCII values (118, 97, 120) of these character constants.

(c) At times we may want to execute a common set of statements for multiple **cases**. How this can be done is shown in the following example.

```
main( )
{
    char  ch ;

    printf ( "Enter any of the alphabet a, b, or c " ) ;
    scanf ( "%c", &ch ) ;
```

```
    switch ( ch )
    {
        case 'a' :
        case 'A' :
            printf ( "a as in ashar" ) ;
            break ;
        case 'b' :
        case 'B' :
            printf ( "b as in brain" ) ;
            break ;
        case 'c' :
        case 'C' :
            printf ( "c as in cookie" ) ;
            break ;
        default :
            printf ( "wish you knew what are alphabets" ) ;
    }
}
```

Here, we are making use of the fact that once a **case** is satisfied the control simply falls through the **case** till it doesn't encounter a **break** statement. That is why if an alphabet **a** is entered the **case 'a'** is satisfied and since there are no statements to be executed in this **case** the control automatically reaches the next **case** i.e. **case 'A'** and executes all the statements in this **case**.

(d) Even if there are multiple statements to be executed in each **case** there is no need to enclose them within a pair of braces (unlike **if**, and **else**).

(e) Every statement in a **switch** must belong to some **case** or the other. If a statement doesn't belong to any **case** the compiler won't report an error. However, the statement would never get executed. For example, in the following program the **printf( )** never goes to work.

```
main( )
{
    int i, j ;

    printf ( "Enter value of i" ) ;
    scanf ( "%d", &i ) ;

    switch ( i )
    {
        printf ( "Hello" ) ;
        case 1 :
            j = 10 ;
            break ;
        case 2 :
            j = 20 ;
            break ;
    }
}
```

(f) If we have no **default** case, then the program simply falls through the entire **switch** and continues with the next instruction (if any,) that follows the closing brace of **switch**.

(g) Is **switch** a replacement for **if**? Yes and no. Yes, because it offers a better way of writing programs as compared to **if**, and no because in certain situations we are left with no choice but to use **if**. The disadvantage of **switch** is that one cannot have a case in a **switch** which looks like:

```
case i <= 20 :
```

All that we can have after the case is an **int** constant or a **char** constant or an expression that evaluates to one of these constants. Even a **float** is not allowed.

The advantage of **switch** over **if** is that it leads to a more structured program and the level of indentation is manageable,

more so if there are multiple statements within each **case** of a **switch**.

(h) We can check the value of any expression in a **switch**. Thus the following **switch** statements are legal.

```
switch ( i + j * k )
switch ( 23 + 45 % 4 * k )
switch ( a < 4 && b > 7 )
```

Expressions can also be used in cases provided they are constant expressions. Thus **case 3 + 7** is correct, however, **case a + b** is incorrect.

(i) The **break** statement when used in a **switch** takes the control outside the **switch**. However, use of **continue** will not take the control to the beginning of **switch** as one is likely to believe.

(j) In principle, a **switch** may occur within another, but in practice it is rarely done. Such statements would be called nested **switch** statements.

(k) The **switch** statement is very useful while writing menu driven programs. This aspect of **switch** is discussed in the exercise at the end of this chapter.

switch Versus *if-else* Ladder

There are some things that you simply cannot do with a **switch**. These are:

(a) A float expression cannot be tested using a **switch**
(b) Cases can never have variable expressions (for example it is wrong to say **case a +3 :**)
(c) Multiple cases cannot use same expressions. Thus the following **switch** is illegal:

```
switch ( a )
{
    case 3 :
        ...
    case 1 + 2 :
        ...
}
```

(a), (b) and (c) above may lead you to believe that these are obvious disadvantages with a **switch**, especially since there weren't any such limitations with **if-else**. Then why use a **switch** at all? For speed—**switch** works faster than an equivalent **if-else** ladder. How come? This is because the compiler generates a jump table for a **switch** during compilation. As a result, during execution it simply refers the jump table to decide which case should be executed, rather than actually checking which case is satisfied. As against this, **if-elses** are slower because they are evaluated at execution time. A **switch** with 10 cases would work faster than an equivalent **if-else** ladder. Also, a **switch** with 2 **cases** would work slower than **if-else** ladder. Why? If the 10th **case** is satisfied then jump table would be referred and statements for the 10th **case** would be executed. As against this, in an **if-else** ladder 10 conditions would be evaluated at execution time, which makes it slow. Note that a lookup in the jump table is faster than evaluation of a condition, especially if the condition is complex.

If on the other hand the conditions in the **if-else** were simple and less in number then **if-else** would work out faster than the lookup mechanism of a **switch**. Hence a **switch** with two **cases** would work slower than an equivalent **if-else**. Thus, you as a programmer should take a decision which of the two should be used when.

The *goto* Keyword

Avoid **goto** keyword! They make a C programmer's life miserable. There is seldom a legitimate reason for using **goto**, and its use is

one of the reasons that programs become unreliable, unreadable, and hard to debug. And yet many programmers find **goto** seductive.

In a difficult programming situation it seems so easy to use a **goto** to take the control where you want. However, almost always, there is a more elegant way of writing the same program using **if**, **for**, **while** and **switch**. These constructs are far more logical and easy to understand.

The big problem with **goto**s is that when we do use them we can never be sure how we got to a certain point in our code. They obscure the flow of control. So as far as possible skip them. You can always get the job done without them. Trust me, with good programming skills **goto** can always be avoided. This is the first and last time that we are going to use **goto** in this book. However, for sake of completeness of the book, the following program shows how to use **goto**.

```
main( )
{
    int goals ;

    printf ( "Enter the number of goals scored against India" ) ;
    scanf ( "%d", &goals ) ;

    if ( goals <= 5 )
        goto sos ;
    else
    {
        printf ( "About time soccer players learnt C\n" ) ;
        printf ( "and said goodbye! adieu! to soccer" ) ;
        exit( ) ; /* terminates program execution */
    }

    sos :
        printf ( "To err is human!" ) ;
```

```
}
```

And here are two sample runs of the program...

```
Enter the number of goals scored against India 3
To err is human!
Enter the number of goals scored against India 7
About time soccer players learnt C
and said goodbye! adieu! to soccer
```

A few remarks about the program would make the things clearer.

- If the condition is satisfied the **goto** statement transfers control to the label 'sos', causing **printf( )** following **sos** to be executed.

- The label can be on a separate line or on the same line as the statement following it, as in,

```
sos : printf ( "To err is human!" ) ;
```

- Any number of **gotos** can take the control to the same label.

- The **exit( )** function is a standard library function which terminates the execution of the program. It is necessary to use this function since we don't want the statement

```
printf ( "To err is human!" )
```

  to get executed after execution of the **else** block.

- The only programming situation in favour of using **goto** is when we want to take the control out of the loop that is contained in several other loops. The following program illustrates this.

```
main( )
{
    int i, j, k ;

    for ( i = 1 ; i <= 3 ; i++ )
    {
        for ( j = 1 ; j <= 3 ; j++ )
        {
            for ( k = 1 ; k <= 3 ; k++ )
            {
                if ( i == 3 && j == 3 && k == 3 )
                    goto out ;
                else
                    printf ( "%d %d %d\n", i, j, k ) ;
            }
        }
    }
    out :
        printf ( "Out of the loop at last!" ) ;
}
```

Go through the program carefully and find out how it works. Also write down the same program without using **goto**.

Summary

(a) When we need to choose one among number of alternatives, a **switch** statement is used.

(b) The **switch** keyword is followed by an integer or an expression that evaluates to an integer.

(c) The **case** keyword is followed by an integer or a character constant.

(d) The control falls through all the cases unless the **break** statement is given.

(e) The usage of the **goto** keyword should be avoided as it usually violets the normal flow of execution.

Exercise

[A] What would be the output of the following programs:

(a)
```
main( )
{
    char  suite = 3 ;
    switch ( suite )
    {
        case 1 :
            printf ( "\nDiamond" ) ;
        case 2 :
            printf ( "\nSpade" ) ;
        default :
            printf ( "\nHeart") ;
    }
    printf ( "\nI thought one wears a suite" ) ;
}
```

(b)
```
main( )
{
    int  c = 3 ;

    switch ( c )
    {
        case 'v' :
            printf ( "I am in case v \n" ) ;
            break ;
        case 3 :
            printf ( "I am in case 3 \n" ) ;
            break ;
        case 12 :
            printf ( "I am in case 12 \n" ) ;
            break ;
        default :
            printf ( "I am in default \n" ) ;
```

```
    }
```

(c)
```
main( )
{
    int k, j = 2 ;
    switch ( k = j + 1 )
    {
        case 0 :
            printf ( "\nTailor") ;
        case 1 :
            printf ( "\nTutor") ;
        case 2 :
            printf ( "\nTramp") ;
        default :
            printf ( "\nPure Simple Egghead!" ) ;
    }
}
```

(d)
```
main( )
{
    int i = 0 ;
    switch ( i )
    {
        case 0 :
            printf ( "\nCustomers are dicey" ) ;
        case 1 :
            printf ( "\nMarkets are pricey" ) ;
        case 2 :
            printf ( "\nInvestors are moody" ) ;
        case 3 :
            printf ( "\nAt least employees are good" ) ;
    }
}
```

(e)
```
main( )
{
    int k ;
    float j = 2.0 ;
```

```
    switch ( k = j + 1 )
    {
        case 3 :
            printf ( "\nTrapped" ) ;
            break ;
        default :
            printf ( "\nCaught!" ) ;
    }
}
```

(f)
```
main( )
{
    int ch = 'a' + 'b' ;
    switch ( ch )
    {
        case 'a' :
        case 'b' :
            printf ( "\nYou entered b" ) ;
        case 'A' :
            printf ( "\na as in ashar" ) ;
        case 'b' + 'a' :
            printf ( "\nYou entered a and b" ) ;
    }
}
```

(g)
```
main( )
{
    int i = 1 ;
    switch ( i - 2 )
    {
        case -1 :
            printf ( "\nFeeding fish" ) ;
        case 0 :
            printf ( "\nWeeding grass" ) ;
        case 1 :
            printf ( "\nmending roof" ) ;
        default :
            printf ( "\nJust to survive" ) ;
```

```
    }
}
```

[B] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    int suite = 1 ;
    switch ( suite ) ;
    {
        case 0 ;
            printf ( "\nClub" ) ;
        case 1 ;
            printf ( "\nDiamond" ) ;
    }
}
```

(b)
```
main( )
{
    int temp ;
    scanf ( "%d", &temp ) ;
    switch ( temp )
    {
        case ( temp <= 20 ) :
            printf ( "\nOoooooohhh! Damn cool!" ) ;
        case ( temp > 20 && temp <= 30 ) :
            printf ( "\nRain rain here again!" ) ;
        case ( temp > 30 && temp <= 40 ) :
            printf ( "\nWish I am on Everest" ) ;
        default :
            printf ( "\nGood old nagpur weather" ) ;
    }
}
```

(c)
```
main( )
{
    float a = 3.5 ;
    switch ( a )
```

```
    {
        case 0.5 :
            printf ( "\nThe art of C" ) ;
            break ;
        case 1.5 :
            printf ( "\nThe spirit of C" ) ;
            break ;
        case 2.5 :
            printf ( "\nSee through C" ) ;
            break ;
        case 3.5 :
            printf ( "\nSimply c" ) ;
    }
}
```

(d)
```
main( )
{
    int a = 3, b = 4, c ;
    c = b - a ;
    switch ( c )
    {
        case 1 || 2 :
            printf ( "God give me an opportunity to change things" ) ;
            break ;

        case a || b :
            printf ( "God give me an opportunity to run my show" ) ;
            break ;
    }
}
```

[C] Write a menu driven program which has following options:

1. Factorial of a number.
2. Prime or not
3. Odd or even
4. Exit

Make use of *switch* statement.

The outline of this program is given below:

```
/* A menu driven program */
main( )
{
    int choice ;
    while ( 1 )
    {
        printf ( "\n1. Factorial" ) ;
        printf ( "\n2. Prime" ) ;
        printf ( "\n3. Odd/Even" ) ;
        printf ( "\n4. Exit" ) ;
        printf ( "\nYour choice? " ) ;
        scanf ( "%d", &choice ) ;

        switch ( choice )
        {
            case 1 :
                /* logic for factorial of a number */
                break ;
            case 2 :
                /* logic for deciding prime number */
                break ;
            case 3 :
                /* logic for odd/even */
                break ;
            case 4 :
                exit( ) ;
        }
    }
}
```

Note:

The statement **while (1)** puts the entire logic in an infinite loop. This is necessary since the menu must keep reappearing on the screen once an item is selected and an appropriate action taken.

[D] Write a program which to find the grace marks for a student using **switch**. The user should enter the class obtained by the student and the number of subjects he has failed in.

- If the student gets first class and the number of subjects he failed in is greater than 3, then he does not get any grace. If the number of subjects he failed in is less than or equal to 3 then the grace is of 5 marks per subject.

- If the student gets second class and the number of subjects he failed in is greater than 2, then he does not get any grace. If the number of subjects he failed in is less than or equal to 2 then the grace is of 4 marks per subject.

- If the student gets third class and the number of subjects he failed in is greater than 1, then he does not get any grace. If the number of subjects he failed in is equal to 1 then the grace is of 5 marks per subject

5 Functions & Pointers

- What is a Function
    - Why Use Functions
- Passing Values between Functions
- Scope Rule of Functions
- Calling Convention
- One Dicey Issue
- Advanced Features of Functions
    - Function Declaration and Prototypes
    - Call by Value and Call by Reference
    - An Introduction to Pointers
    - Pointer Notation
    - Back to Function Calls
    - Conclusions
    - Recursion
- Adding Functions to the Library
- Summary
- Exercise

Knowingly or unknowingly we rely on so many persons for so many things. Man is an intelligent species, but still cannot perform all of life's tasks all alone. He has to rely on others. You may call a mechanic to fix up your bike, hire a gardener to mow your lawn, or rely on a store to supply you groceries every month. A computer program (except for the simplest one) finds itself in a similar situation. It cannot handle all the tasks by itself. Instead, it requests other program like entities—called 'functions' in C—to get its tasks done. In this chapter we will study these functions. We will look at a variety of features of these functions, starting with the simplest one and then working towards those that demonstrate the power of C functions.

What is a Function

A function is a self-contained block of statements that perform a coherent task of some kind. Every C program can be thought of as a collection of these functions. As we noted earlier, using a function is something like hiring a person to do a specific job for you. Sometimes the interaction with this person is very simple; sometimes it's complex.

Suppose you have a task that is always performed exactly in the same way—say a bimonthly servicing of your motorbike. When you want it to be done, you go to the service station and say, "It's time, do it now". You don't need to give instructions, because the mechanic knows his job. You don't need to be told when the job is done. You assume the bike would be serviced in the usual way, the mechanic does it and that's that.

Let us now look at a simple C function that operates in much the same way as the mechanic. Actually, we will be looking at two things—a function that calls or activates the function and the function itself.

```
main( )
{
    message( ) ;
    printf ( "\nCry, and you stop the monotony!" ) ;
}
message( )
{
    printf ( "\nSmile, and the world smiles with you..." ) ;
}
```

And here's the output...

```
Smile, and the world smiles with you...
Cry, and you stop the monotony!
```

Here, **main( )** itself is a function and through it we are calling the function **message( )**. What do we mean when we say that **main( )** 'calls' the function **message( )**? We mean that the control passes to the function **message( )**. The activity of **main( )** is temporarily suspended; it falls asleep while the **message( )** function wakes up and goes to work. When the **message( )** function runs out of statements to execute, the control returns to **main( )**, which comes to life again and begins executing its code at the exact point where it left off. Thus, **main( )** becomes the 'calling' function, whereas **message( )** becomes the 'called' function.

If you have grasped the concept of 'calling' a function you are prepared for a call to more than one function. Consider the following example:

```
main( )
{
    printf ( "\nI am in main" ) ;
    italy( ) ;
    brazil( ) ;
    argentina( ) ;
}
```

```
italy( )
{
    printf ( "\nI am in italy" ) ;

}
brazil( )
{
    printf ( "\nI am in brazil" ) ;
}
argentina( )
{
    printf ( "\nI am in argentina" ) ;
}
```

The output of the above program when executed would be as under:

```
I am in main
I am in italy
I am in brazil
I am in argentina
```

From this program a number of conclusions can be drawn:

- Any C program contains at least one function.
- If a program contains only one function, it must be **main( )**.
- If a C program contains more than one function, then one (and only one) of these functions must be **main( )**, because program execution always begins with **main( )**.
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls in **main( )**.

- After each function has done its thing, control returns to **main( )**. When **main( )** runs out of function calls, the program ends.

As we have noted earlier the program execution always begins with **main( )**. Except for this fact all C functions enjoy a state of perfect equality. No precedence, no priorities, nobody is nobody's boss. One function can call another function it has already called but has in the meantime left temporarily in order to call a third function which will sometime later call the function that has called it, if you understand what I mean. No? Well, let's illustrate with an example.

```
main( )
{
    printf ( "\nI am in main" ) ;
    italy( ) ;
    printf ( "\nI am finally back in main" ) ;
}
italy( )
{
    printf ( "\nI am in italy" ) ;
    brazil( ) ;
    printf ( "\nI am back in italy" ) ;
}
brazil( )
{
    printf ( "\nI am in brazil" ) ;
    argentina( ) ;
}
argentina( )
{
    printf ( "\nI am in argentina" ) ;
}
```

And the output would look like...

```
I am in main
I am in italy
I am in brazil
I am in argentina
I am back in italy
I am finally back in main
```

Here, **main( )** calls other functions, which in turn call still other functions. Trace carefully the way control passes from one function to another. Since the compiler always begins the program execution with **main( )**, every function in a program must be called directly or indirectly by **main( )**. In other words, the **main( )** function drives other functions.

Let us now summarize what we have learnt so far.

(a) C program is a collection of one or more functions.

(b) A function gets called when the function name is followed by a semicolon. For example,

```
main( )
{
    argentina( ) ;
}
```

(c) A function is defined when function name is followed by a pair of braces in which one or more statements may be present. For example,

```
argentina( )
{
    statement 1 ;
    statement 2 ;
    statement 3 ;
}
```

(d) Any function can be called from any other function. Even **main( )** can be called from other functions. For example,

```
main( )
{
    message( ) ;
}
message( )
{
    printf ( "\nCan't imagine life without C" ) ;
    main( ) ;

}
```

(e) A function can be called any number of times. For example,

```
main( )
{
    message( ) ;
    message( ) ;
}
message( )
{
    printf ( "\nJewel Thief!!" ) ;
}
```

(f) The order in which the functions are defined in a program and the order in which they get called need not necessarily be same. For example,

```
main( )
{
    message1( ) ;
    message2( ) ;
}
message2( )
{
    printf ( "\nBut the butter was bitter" ) ;
```

```
}
message1( )
{
    printf ( "\nMary bought some butter" ) ;
}
```

Here, even though **message1( )** is getting called before **message2( )**, still, **message1( )** has been defined after **message2( )**. However, it is advisable to define the functions in the same order in which they are called. This makes the program easier to understand.

(g) A function can call itself. Such a process is called 'recursion'. We would discuss this aspect of C functions later in this chapter.

(h) A function can be called from other function, but a function cannot be defined in another function. Thus, the following program code would be wrong, since **argentina( )** is being defined inside another function, **main( )**.

```
main( )
{
    printf ( "\nI am in main" ) ;
    argentina( )
    {
        printf ( "\nI am in argentina" ) ;
    }
}
```

(i) There are basically two types of functions:

Library functions Ex. **printf( )**, **scanf( )** etc.
User-defined functions Ex. **argentina( )**, **brazil( )** etc.

As the name suggests, library functions are nothing but commonly required functions grouped together and stored in

what is called a Library. This library of functions is present on the disk and is written for us by people who write compilers for us. Almost always a compiler comes with a library of standard functions. The procedure of calling both types of functions is exactly same.

Why Use Functions

Why write separate functions at all? Why not squeeze the entire logic into one function, **main( )**? Two reasons:

(a) Writing functions avoids rewriting the same code over and over. Suppose you have a section of code in your program that calculates area of a triangle. If later in the program you want to calculate the area of a different triangle, you won't like it if you are required to write the same instructions all over again. Instead, you would prefer to jump to a 'section of code' that calculates area and then jump back to the place from where you left off. This section of code is nothing but a function.

(b) Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

What is the moral of the story? Don't try to cram the entire logic in one function. It is a very bad style of programming. Instead, break a program into small units and write functions for each of these isolated subdivisions. Don't hesitate to write functions that are called only once. What is important is that these functions perform some logically isolated task.

Passing Values between Functions

The functions that we have used so far haven't been very flexible. We call them and they do what they are designed to do. Like our mechanic who always services the motorbike in exactly the same way, we haven't been able to influence the functions in the way they carry out their tasks. It would be nice to have a little more control over what functions do, in the same way it would be nice to be able to tell the mechanic, "Also change the engine oil, I am going for an outing". In short, now we want to communicate between the 'calling' and the 'called' functions.

The mechanism used to convey information to the function is the 'argument'. You have unknowingly used the arguments in the **printf( )** and **scanf( )** functions; the format string and the list of variables used inside the parentheses in these functions are arguments. The arguments are sometimes also called 'parameters'.

Consider the following program. In this program, in **main( )** we receive the values of **a**, **b** and **c** through the keyboard and then output the sum of **a**, **b** and **c**. However, the calculation of sum is done in a different function called **calsum( )**. If sum is to be calculated in **calsum( )** and values of **a**, **b** and **c** are received in **main( )**, then we must pass on these values to **calsum( )**, and once **calsum( )** calculates the sum we must return it from **calsum( )** back to **main( )**.

```
/* Sending and receiving values between functions */
main( )
{
    int a, b, c, sum ;

    printf ( "\nEnter any three numbers " ) ;
    scanf ( "%d %d %d", &a, &b, &c ) ;

    sum = calsum ( a, b, c ) ;
```

```
    printf ( "\nSum = %d", sum ) ;
}

calsum ( x, y, z )
int x, y, z ;
{
    int d ;

    d = x + y + z ;
    return ( d ) ;
}
```

And here is the output...

```
Enter any three numbers 10 20 30
Sum = 60
```

There are a number of things to note about this program:

(a) In this program, from the function **main( )** the values of **a, b** and **c** are passed on to the function **calsum( )**, by making a call to the function **calsum( )** and mentioning **a**, **b** and **c** in the parentheses:

```
sum = calsum ( a, b, c ) ;
```

In the **calsum( )** function these values get collected in three variables **x, y** and **z**:

```
calsum ( x, y, z )
int x, y, z ;
```

(b) The variables **a**, **b** and **c** are called 'actual arguments', whereas the variables **x**, **y** and **z** are called 'formal arguments'. Any number of arguments can be passed to a function being called. However, the type, order and number of the actual and formal arguments must always be same.

Instead of using different variable names **x**, **y** and **z**, we could have used the same variable names **a**, **b** and **c**. But the compiler would still treat them as different variables since they are in different functions.

(c) There are two methods of declaring the formal arguments. The one that we have used in our program is known as Kernighan and Ritchie (or just K & R) method.

```
calsum ( x, y, z )
int x, y, z ;
```

Another method is,

```
calsum ( int x, int y, int z )
```

This method is called ANSI method and is more commonly used these days.

(d) In the earlier programs the moment closing brace (}) of the called function was encountered the control returned to the calling function. No separate **return** statement was necessary to send back the control.

This approach is fine if the called function is not going to return any meaningful value to the calling function. In the above program, however, we want to return the sum of **x**, **y** and **z**. Therefore, it is necessary to use the **return** statement.

The **return** statement serves two purposes:

(1) On executing the **return** statement it immediately transfers the control back to the calling program.
(2) It returns the value present in the parentheses after **return**, to th3e calling program. In the above program the value of sum of three numbers is being returned.

(e) There is no restriction on the number of **return** statements that may be present in a function. Also, the **return** statement need not always be present at the end of the called function. The following program illustrates these facts.

```
fun( )
{
    char ch ;

    printf ( "\nEnter any alphabet " ) ;
    scanf ( "%c", &ch ) ;

    if ( ch >= 65 && ch <= 90 )
        return ( ch ) ;
    else
        return ( ch + 32 ) ;
}
```

In this function different **return** statements will be executed depending on whether **ch** is capital or not.

(f) Whenever the control returns from a function some value is definitely returned. If a meaningful value is returned then it should be accepted in the calling program by equating the called function to some variable. For example,

```
sum = calsum ( a, b, c ) ;
```

(g) All the following are valid **return** statements.

```
return ( a ) ;
return ( 23 ) ;
return ( 12.34 ) ;
return ;
```

In the last statement a garbage value is returned to the calling function since we are not returning any specific value. Note that in this case the parentheses after **return** are dropped.

(h) If we want that a called function should not return any value, in that case, we must mention so by using the keyword **void** as shown below.

```
void display( )
{
    printf ( "\nHeads I win..." ) ;
    printf ( "\nTails you lose" ) ;
}
```

(i) A function can return only one value at a time. Thus, the following statements are invalid.

```
return ( a, b ) ;
return ( x, 12 ) ;
```

There is a way to get around this limitation, which would be discussed later in this chapter when we learn pointers.

(j) If the value of a formal argument is changed in the called function, the corresponding change does not take place in the calling function. For example,

```
main( )
{
    int  a = 30 ;
    fun ( a ) ;
    printf ( "\n%d", a ) ;
}

fun ( int  b )
{
    b = 60 ;
```

```
    printf ( "\n%d", b ) ;
}
```

The output of the above program would be:

```
60
30
```

Thus, even though the value of **b** is changed in **fun( )**, the value of **a** in **main( )** remains unchanged. This means that when values are passed to a called function the values present in actual arguments are not physically moved to the formal arguments; just a photocopy of values in actual argument is made into formal arguments.

Scope Rule of Functions

Look at the following program

```
main( )
{
    int i = 20 ;
    display ( i ) ;
}

display ( int j )
{
    int k = 35 ;
    printf ( "\n%d", j ) ;
    printf ( "\n%d", k ) ;
}
```

In this program is it necessary to pass the value of the variable **i** to the function **display( )**? Will it not become automatically available to the function **display( )**? No. Because by default the scope of a variable is local to the function in which it is defined. The presence

of **i** is known only to the function **main( )** and not to any other function. Similarly, the variable **k** is local to the function **display( )** and hence it is not available to **main( )**. That is why to make the value of **i** available to **display( )** we have to explicitly pass it to **display( )**. Likewise, if we want **k** to be available to **main( )** we will have to return it to **main( )** using the **return** statement. In general we can say that the scope of a variable is local to the function in which it is defined..

Calling Convention

Calling convention indicates the order in which arguments are passed to a function when a function call is encountered. There are two possibilities here:

(a) Arguments might be passed from left to right.
(b) Arguments might be passed from right to left.

C language follows the second order.

Consider the following function call:

```
fun (a, b, c, d ) ;
```

In this call it doesn't matter whether the arguments are passed from left to right or from right to left. However, in some function call the order of passing arguments becomes an important consideration. For example:

```
int a = 1 ;
printf ( "%d %d %d", a, ++a, a++ ) ;
```

It appears that this **printf( )** would output 1 2 3.

This however is not the case. Surprisingly, it outputs 3 3 1. This is because C's calling convention is from right to left. That is, firstly

1 is passed through the expression **a++** and then **a** is incremented to 2. Then result of **++a** is passed. That is, **a** is incremented to 3 and then passed. Finally, latest value of **a**, i.e. 3, is passed. Thus in right to left order 1, 3, 3 get passed. Once **printf( )** collects them it prints them in the order in which we have asked it to get them printed (and not the order in which they were passed). Thus 3 3 1 gets printed.

One Dicey Issue

Consider the following function calls:

```
#include <conio.h>
clrscr ( ) ;
gotoxy ( 10, 20 ) ;
ch = getch ( a ) ;
```

Here we are calling three standard library functions. Whenever we call the library functions we must write their prototype before making the call. This helps the compiler in checking whether the values being passed and returned are as per the prototype declaration. But since we don't define the library functions (we merely call them) we may not know the prototypes of library functions. Hence when the library of functions is provided a set of '.h' files is also provided. These files contain the prototypes of library functions. But why multiple files? Because the library functions are divided into different groups and one file is provided for each group. For example, prototypes of all input/output functions are provided in the file 'stdio.h', prototypes of all mathematical functions are provided in the file 'math.h', etc.

On compilation of the above code the compiler reports all errors due to the mismatch between parameters in function call and their corresponding prototypes declared in the file 'conio.h'. You can even open this file and look at the prototypes. They would appear as shown below:

```
void clrscr( ) ;
void gotoxy ( int, int ) ;
int getch( ) ;
```

Now consider the following function calls:

```
#include <stdio.h>
int i = 10, j = 20 ;

printf ( "%d %d %d ", i, j ) ;
printf ( "%d", i, j ) ;
```

The above functions get successfully compiled even though there is a mismatch in the format specifiers and the variables in the list. This is because **printf( )** accepts *variable* number of arguments (sometimes 2 arguments, sometimes 3 arguments, etc.), and even with the mismatch above the call still matches with the prototype of **printf( )** present in 'stdio.h'. At run-time when the first **printf( )** is executed, since there is no variable matching with the last specifier **%d**, a garbage integer gets printed. Similarly, in the second **printf( )** since the format specifier for **j** has not been mentioned its value does not get printed.

Advanced Features of Functions

With a sound basis of the preliminaries of C functions, let us now get into their intricacies. Following advanced topics would be considered here.

(a) Function Declaration and Prototypes
(b) Calling functions by value or by reference
(c) Recursion

Let us understand these features one by one.

Function Declaration and Prototypes

Any C function by default returns an **int** value. More specifically, whenever a call is made to a function, the compiler assumes that this function would return a value of the type **int**. If we desire that a function should return a value other than an **int**, then it is necessary to explicitly mention so in the calling function as well as in the called function. Suppose we want to find out square of a number using a function. This is how this simple program would look like:

```
main( )
{
    float a, b ;

    printf ( "\nEnter any number " ) ;
    scanf ( "%f", &a ) ;

    b = square ( a ) ;
    printf ( "\nSquare of %f is %f", a, b ) ;
}

square ( float x )
{
    float y ;

    y = x * x ;
    return ( y ) ;
}
```

And here are three sample runs of this program...

```
Enter any number 3
Square of 3 is 9.000000
Enter any number 1.5
Square of 1.5 is 2.000000
Enter any number 2.5
Square of 2.5 is 6.000000
```

The first of these answers is correct. But square of 1.5 is definitely not 2. Neither is 6 a square of 2.5. This happened because any C function, by default, always returns an integer value. Therefore, even though the function **square( )** calculates the square of 1.5 as 2.25, the problem crops up when this 2.25 is to be returned to **main( )**. **square( )** is not capable of returning a **float** value. How do we overcome this? The following program segment illustrates how to make **square( )** capable of returning a **float** value.

```
main( )
{
    float square ( float ) ;
    float a, b ;

    printf ( "\nEnter any number " ) ;
    scanf ( "%f", &a ) ;

    b = square ( a ) ;
    printf ( "\nSquare of %f is %f", a, b ) ;
}

float square ( float x )
{
    float y ;
    y = x * x ;
    return ( y ) ;
}
```

And here is the output...

```
Enter any number 1.5
Square of 1.5 is 2.250000
Enter any number 2.5
Square of 2.5 is 6.250000
```

Now the expected answers i.e. 2.25 and 6.25 are obtained. Note that the function **square( )** must be declared in **main( )** as

```
float square ( float ) ;
```

This statement is often called the prototype declaration of the **square( )** function. What it means is **square( )** is a function that receives a **float** and returns a **float**. We have done the prototype declaration in **main( )** because we have called it from **main( )**. There is a possibility that we may call **square( )** from several other functions other than **main( )**. Does this mean that we would need prototype declaration of **square( )** in all these functions. No, in such a case we would make only one declaration outside all the functions at the beginning of the program.

In practice you may seldom be required to return a value other than an **int**, but just in case you are required to, employ the above method. In some programming situations we want that a called function should not return any value. This is made possible by using the keyword **void**. This is illustrated in the following program.

```
main( )
{
    void gospel( ) ;
    gospel( ) ;
}

void gospel( )
{
    printf ( "\nViruses are electronic bandits..." ) ;
    printf ( "\nwho eat nuggets of information..." ) ;
    printf ( "\nand chunks of bytes..." ) ;
    printf ( "\nwhen you least expect..." ) ;
}
```

Here, the **gospel( )** function has been defined to return **void**; means it would return nothing. Therefore, it would just flash the four messages about viruses and return the control back to the **main( )** function.

Call by Value and Call by Reference

By now we are well familiar with how to call functions. But, if you observe carefully, whenever we called a function and passed something to it we have always passed the 'values' of variables to the called function. Such function calls are called 'calls by value'. By this what we mean is, on calling a function we are passing values of variables to it. The examples of call by value are shown below:

```
sum = calsum ( a, b, c ) ;
f = factr ( a ) ;
```

We have also learnt that variables are stored somewhere in memory. So instead of passing the value of a variable, can we not pass the location number (also called address) of the variable to a function? If we were able to do so it would become a 'call by reference'. What purpose a 'call by reference' serves we would find out a little later. First we must equip ourselves with knowledge of how to make a 'call by reference'. This feature of C functions needs at least an elementary knowledge of a concept called 'pointers'. So let us first acquire the basics of pointers after which we would take up this topic once again.

An Introduction to Pointers

Which feature of C do beginners find most difficult to understand? The answer is easy: pointers. Other languages have pointers but few use them so frequently as C does. And why not? It is C's clever use of pointers that makes it the excellent language it is.

The difficulty beginners have with pointers has much to do with C's pointer terminology than the actual concept. For instance, when a C programmer says that a certain variable is a "pointer", what does that mean? It is hard to see how a variable can point to something, or in a certain direction.

It is hard to get a grip on pointers just by listening to programmer's jargon. In our discussion of C pointers, therefore, we will try to avoid this difficulty by explaining pointers in terms of programming concepts we already understand. The first thing we want to do is explain the rationale of C's pointer notation.

Pointer Notation

Consider the declaration,

```
int i = 3 ;
```

This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.
(b) Associate the name i with this memory location.
(c) Store the value 3 at this location.

We may represent i's location in memory by the following memory map.

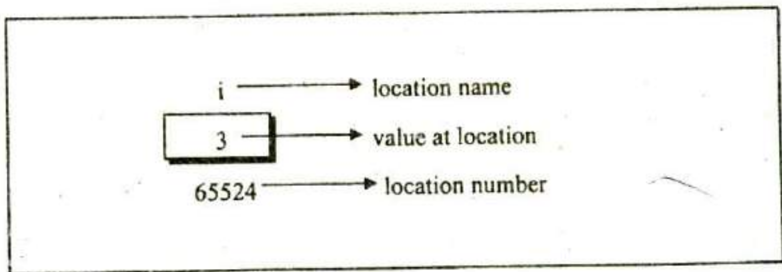


Figure 5.1

We see that the computer has selected memory location 65524 as the place to store the value 3. The location number 65524 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. The important point is, **i**'s address in memory is a number.

We can print this address number through the following program:

```
main( )
{
    int i = 3 ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nValue of i = %d", i ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
```

Look at the first **printf( )** statement carefully. '&' used in this statement is C's 'address of' operator. The expression **&i** returns the address of the variable **i**, which in this case happens to be 65524. Since 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using **%u**, which is a format specifier for printing an unsigned integer. We have been using the '&' operator all the time in the **scanf( )** statement.

The other pointer operator available in C is '*', called 'value at address' operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.

Observe carefully the output of the following program:

```
main( )
{
    int i = 3 ;

    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
Value of i = 3
```

Note that printing the value of ***(&i)** is same as printing the value of **i**.

The expression **&i** gives the address of the variable **i**. This address can be collected in a variable, by saying,

```
j = &i ;
```

But remember that **j** is not an ordinary variable like any other integer variable. It is a variable that contains the address of other variable (**i** in this case). Since **j** is a variable the compiler must provide it space in the memory. Once again, the following memory map would illustrate the contents of **i** and **j**.

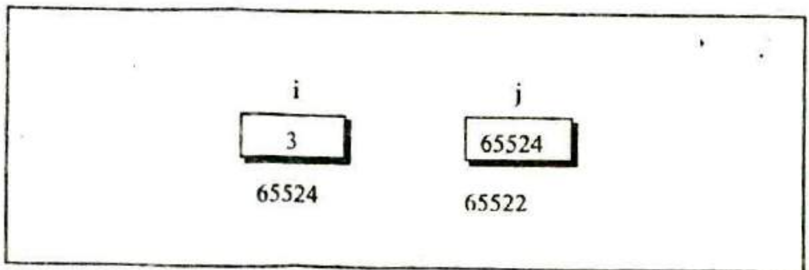


Figure 5.2

As you can see, **i**'s value is 3 and **j**'s value is **i**'s address.

But wait, we can't use **j** in a program without declaring it. And since **j** is a variable that contains the address of **i**, it is declared as,

```
int *j ;
```

This declaration tells the compiler that **j** will be used to store the address of an integer value. In other words **j** points to an integer. How do we justify the usage of * in the declaration,

```
int *j ;
```

Let us go by the meaning of *. It stands for 'value at address'. Thus, **int *j** would mean, the value at the address contained in **j** is an **int**.

Here is a program that demonstrates the relationships we have been discussing.

```
main( )
{
    int i = 3 ;
    int *j ;

    j = &i ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nAddress of i = %u", j ) ;
    printf ( "\nAddress of j = %u", &j ) ;
    printf ( "\nValue of j = %u", j ) ;
    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of i = %d", *( &i ) ) ;
    printf ( "\nValue of i = %d", *j ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3
```

Work through the above program carefully, taking help of the memory locations of **i** and **j** shown earlier. This program summarizes everything that we have discussed so far. If you don't understand the program's output, or the meanings of **&i**, **&j**, ***j** and ***(&i)**, re-read the last few pages. Everything we say about C pointers from here onwards will depend on your understanding these expressions thoroughly.

Look at the following declarations,

```
int *alpha ;
char *ch ;
float *s ;
```

Here, **alpha, ch** and **s** are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers, therefore pointers always contain whole numbers. Now we can put these two facts together and say—pointers are variables that contain addresses, and since addresses are always whole numbers, pointers would always contain whole numbers.

The declaration **float *s** does not mean that s is going to contain a floating-point value. What it means is, **s** is going to contain the address of a floating-point value. Similarly, **char *ch** means that **ch** is going to contain the address of a char value. Or in other words, the value at address stored in **ch** is going to be a **char**.

The concept of pointers can be further extended. Pointer, we know is a variable that contains address of another variable. Now this variable itself might be another pointer. Thus, we now have a pointer that contains another pointer's address. The following example should make this point clear.

```
main( )
{
    int i = 3, *j, **k ;

    j = &i ;
    k = &j ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nAddress of i = %u", j ) ;
    printf ( "\nAddress of i = %u", *k ) ;
    printf ( "\nAddress of j = %u", &j ) ;
    printf ( "\nAddress of j = %u", k ) ;
    printf ( "\nAddress of k = %u", &k ) ;
    printf ( "\nValue of j  = %u", j ) ;
    printf ( "\nValue of k  = %u", k ) ;
    printf ( "\nValue of i  = %d", i ) ;
    printf ( "\nValue of i  = %d", * ( &i ) ) ;
    printf ( "\nValue of i  = %d", *j ) ;
    printf ( "\nValue of i  = %d", **k ) ;
}
```

The output of the above program would be:

```
Address of i = 65524

Address of i = 65524
Address of i = 65524
Address of j = 65522
Address of j = 65522
Address of k = 65520
Value of j  = 65524
Value of k  = 65522
```

```
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3
```

Figure 5.3 would help you in tracing out how the program prints the above output.

Remember that when you run this program the addresses that get printed might turn out to be something different than the ones shown in the figure. However, with these addresses too the relationship between **i**, **j** and **k** can be easily established.

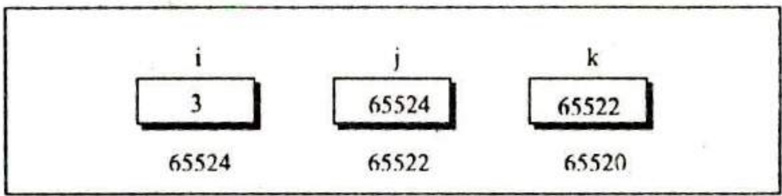


Figure 5.3

Observe how the variables **j** and **k** have been declared,

```
int i, *j, **k ;
```

Here, **i** is an ordinary **int**, **j** is a pointer to an **int** (often called an integer pointer), whereas **k** is a pointer to an integer pointer. We can extend the above program still further by creating a pointer to a pointer to an integer pointer. In principle, you would agree that likewise there could exist a pointer to a pointer to a pointer to a pointer to a pointer. There is no limit on how far can we go on extending this definition. Possibly, till the point we can comprehend it. And that point of comprehension is usually a pointer to a pointer. Beyond this one rarely requires to extend the definition of a pointer. But just in case...

Back to Function Calls

Having had the first tryst with pointers let us now get back to what we had originally set out to learn—the two types of function calls—call by value and call by reference. Arguments can generally be passed to functions in one of the two ways:

(a) sending the values of the arguments
(b) sending the addresses of the arguments

In the first method the 'value' of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function. The following program illustrates the 'Call by Value'.

```
main( )
{
    int a = 10, b = 20 ;

    swapv ( a, b ) ;
    printf ( "\na = %d b = %d", a, b ) ;
}

swapv ( int x, int y )
{
    int t ;

    t = x ;
    x = y ;
    y = t ;

    printf ( "\nx = %d y = %d", x, y ) ;
}
```

The output of the above program would be:

```
x = 20 y = 10
a = 10 b = 20
```

Note that values of **a** and **b** remain unchanged even after exchanging the values of **x** and **y**.

In the second method (call by reference) the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact.

```
main( )
{
    int a = 10, b = 20 ;

    swapr ( &a, &b ) ;
    printf ( "\na = %d b = %d", a, b ) ;
}

swapr( int *x, int *y )
{
    int t ;

    t = *x ;
    *x = *y ;
    *y = t ;
}
```

The output of the above program would be:

```
a = 20 b = 10
```

Note that this program manages to exchange the values of **a and b** using their addresses stored in **x** and **y**.

Usually in C programming we make a call by value. This means that in general you cannot alter the actual arguments. But if desired, it can always be achieved through a call by reference.

Using a call by reference intelligently we can make a function return more than one value at a time, which is not possible ordinarily. This is shown in the program given below.

```
main( )
{
    int radius ;
    float area, perimeter ;

    printf ( "\nEnter radius of a circle " ) ;
    scanf ( "%d", &radius ) ;
    areaperi ( radius, &area, &perimeter ) ;

    printf ( "Area = %f", area ) ;
    printf ( "\nPerimeter = %f", perimeter ) ;
}

areaperi ( int r, float *a, float *p )
{
    *a = 3.14 * r * r ;
    *p = 2 * 3.14 * r ;
}
```

And here is the output...

```
Enter radius of a circle 5
Area = 78.500000
Perimeter = 31.400000
```

Here, we are making a mixed call, in the sense, we are passing the value of **radius** but, addresses of **area** and **perimeter**. And since we are passing the addresses, any change that we make in values stored at addresses contained in the variables **a** and **p**, would make

the change effective in **main( )**. That is why when the control returns from the function **areaperi( )** we are able to output the values of **area** and **perimeter**.

Thus, we have been able to indirectly return two values from a called function, and hence, have overcome the limitation of the **return** statement, which can return only one value from a function at a time.

Conclusions

From the programs that we discussed here we can draw the following conclusions:

(a) If we want that the value of an actual argument should not get changed in the function being called, pass the actual argument by value.

(b) If we want that the value of an actual argument should get changed in the function being called, pass the actual argument by reference.

(c) If a function is to be made to return more than one value at a time then return these values indirectly by using a call by reference.

Recursion

In C, it is possible for the functions to call themselves. A function is called 'recursive' if a statement within the body of a function calls the same function. Sometimes called 'circular definition', recursion is thus the process of defining something in terms of itself.

Let us now see a simple example of recursion. Suppose we want to calculate the factorial value of an integer. As we know, the

factorial of a number is the product of all the integers between 1 and that number. For example, 4 factorial is 4 * 3 * 2 * 1. This can also be expressed as 4! = 4 * 3! where '!' stands for factorial. Thus factorial of a number can be expressed in the form of itself. Hence this can be programmed using recursion. However, before we try to write a recursive function for calculating factorial let us take a look at the non-recursive function for calculating the factorial value of an integer.

```
main( )
{
    int a, fact ;

    printf ( "\nEnter any number " ) ;
    scanf ( "%d", &a ) ;

    fact = factorial ( a ) ;
    printf ( "Factorial value = %d", fact ) ;
}

factorial ( int x )
{
    int f = 1, i ;

    for ( i = x ; i >= 1 ; i-- )
        f = f * i ;

    return ( f ) ;
}
```

And here is the output...

```
Enter any number 3
Factorial value = 6
```

Work through the above program carefully, till you understand the logic of the program properly. Recursive factorial function can be understood only if you are thorough with the above logic.

Following is the recursive version of the function to calculate the factorial value.

```
main( )
{
    int  a, fact ;

    printf ( "\nEnter any number " ) ;
    scanf ( "%d", &a ) ;

    fact = rec ( a ) ;
    printf ( "Factorial value = %d", fact ) ;
}

rec ( int  x )
{
    int  f ;

    if ( x == 1 )
        return ( 1 ) ;
    else
        f = x * rec ( x - 1 ) ;

    return ( f ) ;
}
```

And here is the output for four runs of the program

```
Enter any number 1
Factorial value = 1
Enter any number 2
Factorial value = 2
Enter any number 3
```

```
Factorial value = 6
Enter any number 5
Factorial value = 120
```

Let us understand this recursive factorial function thoroughly. In the first run when the number entered through **scanf( )** is 1, let us see what action does **rec( )** take. The value of **a** (i.e. 1) is copied into **x**. Since **x** turns out to be 1 the condition **if (x == 1)** is satisfied and hence 1 (which indeed is the value of 1 factorial) is returned through the **return** statement.

When the number entered through **scanf( )** is 2, the **(x == 1)** test fails, so we reach the statement,

```
f = x * rec ( x - 1 ) ;
```

And here is where we meet recursion. How do we handle the expression **x * rec (x - 1)**? We multiply **x** by **rec (x - 1)**. Since the current value of **x** is 2, it is same as saying that we must calculate the value (2 * rec (1)). We know that the value returned by **rec (1)** is 1, so the expression reduces to (2 * 1), or simply 2. Thus the statement,

```
x * rec ( x - 1 ) ;
```

evaluates to 2, which is stored in the variable **f**, and is returned to **main( )**, where it is duly printed as

```
Factorial value = 2
```

Now perhaps you can see what would happen if the value of **a** is 3, 4, 5 and so on.

In case the value of **a** is 5, **main( )** would call **rec( )** with 5 as its actual argument, and **rec( )** will send back the computed value. But before sending the computed value, **rec( )** calls **rec( )** and waits for a value to be returned. It is possible for the **rec( )** that has just been

called to call yet another **rec( )**, the argument **x** being decreased in value by 1 for each of these recursive calls. We speak of this series of calls to **rec( )** as being different invocations of **rec( )**. These successive invocations of the same function are possible because the C compiler keeps track of which invocation calls which. These recursive invocations end finally when the last invocation gets an argument value of 1, which the preceding invocation of **rec( )** now uses to calculate its own **f** value and so on up the ladder. So we might say what happens is,

```
rec ( 5 ) returns ( 5 times rec ( 4 ),
    which returns ( 4 times rec ( 3 ),
        which returns ( 3 times rec ( 2 ),
            which returns ( 2 times rec ( 1 ),
                which returns ( 1 ) ) ) ) )
```

Foxed? Well, that is recursion for you in its simplest garbs. I hope you agree that it's difficult to visualize how the control flows from one function call to another. Possibly Figure 5.4 would make things a bit clearer.

Assume that the number entered through **scanf( )** is 3. Using Figure 5.4 let's visualize what exactly happens when the recursive function **rec( )** gets called. Go through the figure carefully. The first time when **rec( )** is called from **main( )**, **x** collects 3. From here, since **x** is not equal to 1, the **if** block is skipped and **rec( )** is called again with the argument **(x – 1)**, i.e. 2. This is a recursive call. Since **x** is still not equal to 1, **rec( )** is called yet another time, with argument (2 - 1). This time as **x** is 1, control goes back to previous **rec( )** with the value 1, and **f** is evaluated as 2.

Similarly, each **rec( )** evaluates its **f** from the returned value, and finally 6 is returned to **main( )**. The sequence would be grasped better by following the arrows shown in Figure 5.4. Let it be clear that while executing the program there do not exist so many copies of the function **rec( )**. These have been shown in the figure just to

help you keep track of how the control flows during successive recursive calls.

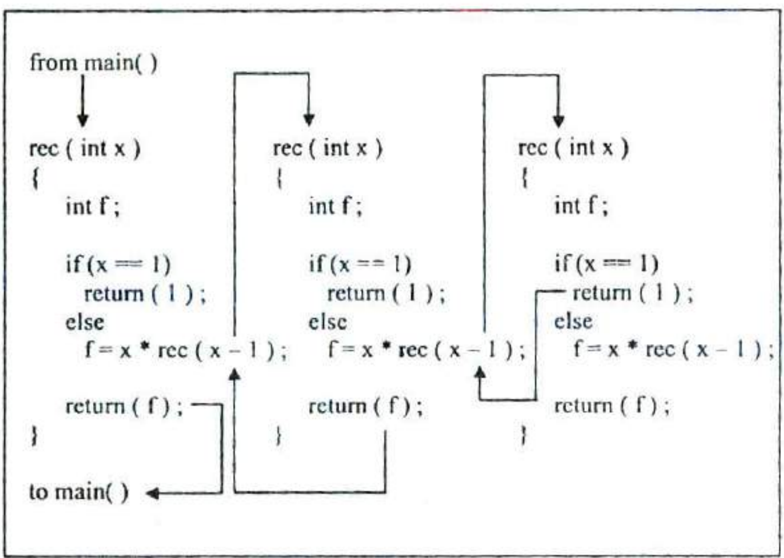


Figure 5.4

Recursion may seem strange and complicated at first glance, but it is often the most direct way to code an algorithm, and once you are familiar with recursion, the clearest way of doing so.

Recursion and Stack

There are different ways in which data can be organized. For example, if you are to store five numbers then we can store them in five different variables, an array, a linked list, a binary tree, etc. All these different ways of organizing the data are known as data structures. The compiler uses one such data structure called stack for implementing normal as well as recursive function calls.

A stack is a Last In First Out (LIFO) data structure. This means that the last item to get stored on the stack (often called Push operation) is the first one to get out of it (often called as Pop operation). You can compare this to the stack of plates in a cafeteria—the last plate that goes on the stack is the first one to get out of it. Now let us see how the stack works in case of the following program.

```
main( )
{
    int a = 5, b = 2, c ;
    c = add ( a, b ) ;
    printf ( "sum = %d", c ) ;
}
add ( int i, int j )
{
    int sum ;
    sum = i + j ;
    return sum ;
}
```

In this program before transferring the execution control to the function **fun( )** the values of parameters **a** and **b** are pushed onto the stack. Following this the address of the statement **printf( )** is pushed on the stack and the control is transferred to **fun( )**. It is necessary to push this address on the stack. In **fun( )** the values of **a** and **b** that were pushed on the stack are referred as **i** and **j**. In **fun( )** the local variable **sum** gets pushed on the stack. When value of **sum** is returned **sum** is popped up from the stack. Next the address of the statement where the control should be returned is popped up from the stack. Using this address the control returns to the **printf( )** statement in **main( )**. Before execution of **printf( )** begins the two integers that were earlier pushed on the stack are now popped off.

How the values are being pushed and popped even though we didn't write any code to do so? Simple—the compiler on

encountering the function call would generate code to push parameters and the address. Similarly, it would generate code to clear the stack when the control returns back from **fun( )**. Figure 5.5 shows the contents of the stack at different stages of execution.

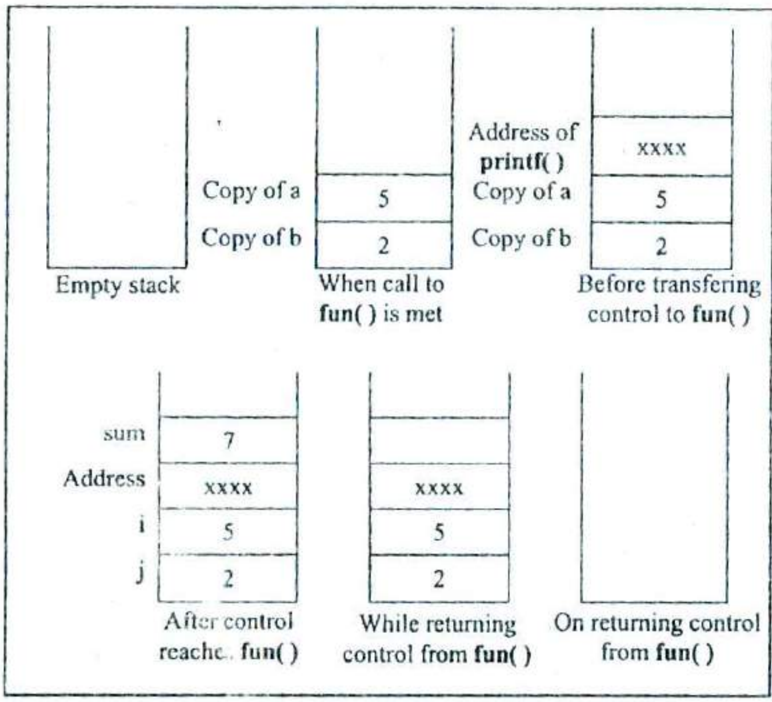


Figure 5.5

Note that in this program popping of **sum** and address is done by **fun( )**, whereas popping of the two integers is done by **main( )**. When it is done this way it is known as 'CDecl Calling Convention'. There are other calling conventions as well where instead of **main( )**, **fun( )** itself clears the two integers. The calling convention also decides whether the parameters being passed to the function are pushed on the stack in left-to-right or right-to-left order. The standard calling convention always uses the right-to-left

order. Thus during the call to **fun( )** firstly value of **b** is pushed to the stack, followed by the value of **a**.

The recursive calls are no different. Whenever we make a recursive call the parameters and the return address gets pushed on the stack. The stack gets unwound when the control returns from the called function. Thus during every recursive function call we are working with a fresh set of parameters.

Also, note that while writing recursive functions you must have an **if** statement somewhere in the recursive function to force the function to return without recursive call being executed. If you don't do this and you call the function, you will fall in an indefinite loop, and the stack will keep on getting filled with parameters and the return address each time there is a call. Soon the stack would become full and you would get a run-time error indicating that the stack has become full. This is a very common error while writing recursive functions. My advice is to use **printf( )** statement liberally during the development of recursive function, so that you can watch what is going on and can abort execution if you see that you have made a mistake.

Adding Functions to the Library

Most of the times we either use the functions present in the standard library or we define our own functions and use them. Can we not add our functions to the standard library? And would it make any sense in doing so? We can add user-defined functions to the library. It makes sense in doing so as the functions that are to be added to the library are first compiled and then added. When we use these functions (by calling them) we save on their compilation time as they are available in the library in the compiled form.

Let us now see how to add user-defined functions to the library. Different compilers provide different utilities to add/delete/modify functions in the standard library. For example, Turbo C/C++

compilers provide a utility called 'tlib.exe' (Turbo Librarian). Let us use this utility to add a function **factorial( )** to the library.

Given below are the steps to do so:

(a) Write the function definition of **factorial( )** in some file, say 'fact.c'.

```
int factorial ( int num )
{
    int i, f = 1 ;
    for ( i = 1 ; i <= num ; i++ )
        f = f * i ;
    return ( f ) ;
}
```

(b) Compile the 'fact.c' file using Alt F9. A new file called 'fact.obj' would get created containing the compiled code in machine language.

(c) Add the function to the library by issuing the command

```
C:\>tlib  math.lib + c:\fact.obj
```

Here, 'math.lib' is a library filename, + is a switch, which means we want to add new function to library and 'c:\fact.obj' is the path of the '.obj' file.

(d) Declare the prototype of the **factorial( )** function in the header file, say 'fact.h'. This file should be included while calling the function.

(e) To use the function present inside the library, create a program as shown below:

```
#include "c:\\fact.h"
main( )
```

```
{
    int f ;
    f = factorial ( 5 ) ;
    printf ( "%d", f ) ;
}
```

(f) Compile and execute the program using Ctrl F9.

If we wish we can delete the existing functions present in the library using the minus (-) switch.

Instead of modifying the existing libraries we can create our own library. Let's see how to do this. Let us assume that we wish to create a library containing the functions **factorial( )**, **prime( )** and **fibonacci( )**. As their names suggest, **factorial( )** calculates and returns the factorial value of the integer passed to it, **prime( )** reports whether the number passed to it is a prime number or not and **fibonacci( )** prints the first **n** terms of the Fibonacci series, where **n** is the number passed to it. Here are the steps that need to be carried out to create this library. Note that these steps are specific to Turbo C/C++ compiler and would vary for other compilers.

(a) Define the functions **factorial( )**, **prime( )** and **fibonacci( )** in a file, say 'myfuncs.c'. Do not define **main( )** in this file.

(b) Create a file 'myfuncs.h' and declare the prototypes of **factorial( )**, **prime( )** and **fibonacci( )** in it as shown below:

```
int factorial ( int ) ;
int prime ( int ) ;
void fibonacci ( int ) ;
```

(c) From the Options menu select the menu-item 'Application'. From the dialog that pops us select the option 'Library'. Select OK.

(d) Compile the program using Alt F9. This would create the library file called 'myfuncs.lib'.

That's it. The library now stands created. Now we have to use the functions defined in this library. Here is how it can be done.

(a) Create a file, say 'sample.c' and type the following code in it.

```
#include "myfuncs.h"
main( )
{
    int f, result ;
    f = factorial ( 5 ) ;
    result = prime ( 13 ) ;
    fibonacci ( 6 ) ;
    printf ( "\n%d %d", f, result ) ;
}
```

Note that the file 'myfuncs.h' should be in the same directory as the file 'sample.c'. If not, then while including 'myfuncs.h' mention the appropriate path.

(b) Go to the 'Project' menu and select 'Open Project...' option. On doing so a dialog would pop up. Give the name of the project, say 'sample.prj' and select OK.

(c) From the 'Project' menu select 'Add Item'. On doing so a file dialog would appear. Select the file 'sample.c' and then select 'Add'. Also add the file 'myfuncs.lib' in the same manner. Finally select 'Done'

(d) Compile and execute the project using Ctrl F9.

Summary

(a) To avoid repetition of code and bulky programs functionally related statements are isolated into a function.

(b) Function declaration specifies what is the return type of the function and the types of parameters it accepts.

(c) Function definition defines the body of the function.

(d) Variables declared in a function are not available to other functions in a program. So, there won't be any clash even if we give same name to the variables declared in different functions.

(e) Pointers are variables which hold addresses of other variables.

(f) A function can be called either by value or by reference.

(g) Pointers can be used to make a function return more than one value simultaneously.

(h) Recursion is difficult to understand, but in some cases offer a better solution than loops.

(i) Adding too many functions and calling them frequently may slow down the program execution.

Exercise

Simple functions, Passing values between functions

[A] What would be the output of the following programs:

(a)
```
main( )
{
    printf ( "\nOnly stupids use C?" ) ;
    display( ) ;
}
display( )
{
    printf ( "\nFools too use C!" ) ;
    main( ) ;
}
```

(b)
```
main( )
{
    printf ( "\nC to it that C survives" ) ;
    main( ) ;
}
```

(c)
```
main( )
{
    int i = 45, c ;
    c = check ( i ) ;
    printf ( "\n%d", c ) ;
}
check ( int ch )
{
    if ( ch >= 45 )
        return ( 100 ) ;
    else
        return ( 10 * 10 ) ;
}
```

(d)
```
main( )
{
    int i = 45, c ;
    c = multiply ( i * 1000 ) ;
    printf ( "\n%d", c ) ;
}
check ( int ch )
{
    if ( ch >= 40000 )
        return ( ch / 10 ) ;
    else
        return ( 10 ) ;
}
```

[B] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
```

```
    int i = 3, j = 4, k, l ;
    k = addmult ( i, j ) ;
    l = addmult ( i, j ) ;
    printf ( "\n%d %d", k, l ) ;
}
addmult ( int ii, int jj )
{
    int kk, ll ;
    kk = ii + jj ;
    ll = ii * jj ;
    return ( kk, ll ) ;
}
```

(b)
```
main( )
{
    int a ;
    a = message( ) ;
}
message( )
{
    printf ( "\nViruses are written in C" ) ;
    return ;
}
```

(c)
```
main( )
{
    float a = 15.5 ;
    char ch = 'C' ;
    printit ( a, ch ) ;
}
printit ( a, ch )
{
    printf ( "\n%f %c", a, ch ) ;
}
```

(d)
```
main( )
{
    message( ) ;
```

```
        message( ) ;
    }
    message( ) ;
    {
        printf ( "\nPraise worthy and C worthy are synonyms" ) ;
    }
```

(e)
```
    main( )
    {
        let_us_c( )
        {
            printf ( "\nC is a Cimple minded language !" ) ;
            printf ( "\nOthers are of course no match !" ) ;
        }
    }
```

(f)
```
    main( )
    {
        message( message ( ) ) ;
    }
    void message( )
    {
        printf ( "\nPraise worthy and C worthy are synonyms" ) ;
    }
```

[C] Answer the following:

(a) Is this a correctly written function:

```
    sqr ( a ) ;
    int a ;
    {
        return ( a * a ) ;
    }
```

(b) State whether the following statements are True or False:

1. The variables commonly used in C functions are available to all the functions in a program.

2. To return the control back to the calling function we must use the keyword **return**.

3. The same variable names can be used in different functions without any conflict.

4. Every called function must contain a **return** statement.

5. A function may contain more than one **return** statements.

6. Each **return** statement in a function may return a different value.

7. A function can still be useful even if you don't pass any arguments to it and the function doesn't return any value back.

8. Same names can be used for different functions without any conflict.

9. A function may be called more than once from any other function.

10. It is necessary for a function to return some value.

[D] Answer the following:

(a) Write a function to calculate the factorial value of any integer entered through the keyboard.

(b) Write a function **power (a, b)**, to calculate the value of **a** raised to **b**.

(c) Write a general-purpose function to convert any given year into its roman equivalent. The following table shows the roman equivalents of decimal numbers:

Decimal	Roman	Decimal	Roman
1	i	100	c
5	v	500	d
10	x	1000	m
50	l		

Example:

Roman equivalent of 1988 is mdcccclxxxviii
Roman equivalent of 1525 is mdxxv

(d) Any year is entered through the keyboard. Write a function to determine whether the year is a leap year or not.

(e) A positive integer is entered through the keyboard. Write a function to obtain the prime factors of this number.

For example, prime factors of 24 are 2, 2, 2 and 3, whereas prime factors of 35 are 5 and 7.

Function Prototypes, Call by Value/Reference, Pointers

[E] What would be the output of the following programs:

(a)
```
main( )
{
    float area ;
    int radius = 1 ;
    area = circle ( radius ) ;
    printf ( "\n%f", area ) ;
}
circle ( int r )
```

```
{
    float a ;
    a = 3.14 * r * r ;
    return ( a ) ;
}
```

(b)
```
main( )
{
    void slogan( ) ;
    int c = 5 ;
    c = slogan( ) ;
    printf ( "\n%d", c ) ;
}
void slogan( )
{
    printf ( "\nOnly He men use C!" ) ;
}
```

[F] Answer the following:

(a) Write a function which receives a **float** and an **int** from **main( )**, finds the product of these two and returns the product which is printed through **main( )**.

(b) Write a function that receives 5 integers and returns the sum, average and standard deviation of these numbers. Call this function from **main( )** and print the results in **main( )**.

(c) Write a function that receives marks received by a student in 3 subjects and returns the average and percentage of these marks. Call this function from **main( )** and print the results in **main( )**.

[G] What would be the output of the following programs:

(a)
```
main( )
{
    int i = 5, j = 2 ;
```

```
    junk ( i, j ) ;
    printf ( "\n%d %d", i, j ) ;
}
junk ( int i, int j )
{
    i = i * i ;
    j = j * j ;
}
```

(b)
```
main( )
{
    int i = 5, j = 2 ;
    junk ( &i, &j ) ;
    printf ( "\n%d %d", i, j ) ;
}
junk ( int *i, int *j )
{
    *i = *i * *i ;
    *j = *j * *j ;
}
```

(c)
```
main( )
{
    int i = 4, j = 2 ;
    junk ( &i, j ) ;
    printf ( "\n%d %d", i, j ) ;
}
junk ( int *i, int j )
{
    *i = *i * *i ;
    j = j * j ;
}
```

(d)
```
main( )
{
    float a = 13.5 ;
    float *b, *c ;
    b = &a ; /* suppose address of a is 1006 */
```

```
    c = b ;
    printf ( "\n%u %u %u", &a, b, c ) ;
    printf ( "\n%f %f %f %f %f", a, *(&a), *&a, *b, *c ) ;
}
```

[H] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    int i = 135, a = 135, k ;
    k = pass ( i, a ) ;
    printf ( "\n%d", k ) ;
}
pass ( int j, int b )
int c ;
{
    c = j + b ;
    return ( c ) ;
}
```

(b)
```
main( )
{
    int p = 23, f = 24 ;
    jiaayjo ( &p, &f ) ;
    printf ( "\n%d %d", p, f ) ;
}
jiaayjo ( int q, int g )
{
    q = q + q ;
    g = g + g ;
}
```

(c)
```
main( )
{
    int k = 35, z ;
    z = check ( k ) ;
    printf ( "\n%d", z ) ;
}
```

```
check ( m )
{
    int m ;
    if ( m > 40 )
        return ( 1 ) ;
    else
        return ( 0 ) ;
}
```

(d)
```
main( )
{
    int i = 35, *z ;
    z = function ( &i ) ;
    printf ( "\n%d", z ) ;
}
function ( int *m )
{
    return ( m + 2 ) ;
}
```

[I] What would be the output of the following programs:

(a)
```
main( )
{
    int i = 0 ;
    i++ ;
    if ( i <= 5 )
    {
        printf ( "\nC adds wings to your thoughts" ) ;
        exit( ) ;
        main( ) ;
    }
}
```

(b)
```
main( )
{
    static int i = 0 ;
    i++ ;
```

```
    if ( i <= 5 )
    {
        printf ( "\n%d", i ) ;
        main( ) ;
    }
    else
        exit( ) ;
}
```

[J] Attempt the following:

(a) A 5-digit positive integer is entered through the keyboard, write a function to calculate sum of digits of the 5-digit number:

(1) Without using recursion
(2) Using recursion

(b) A positive integer is entered through the keyboard, write a program to obtain the prime factors of the number. Modify the function suitably to obtain the prime factors recursively.

(c) Write a recursive function to obtain the first 25 numbers of a Fibonacci sequence. In a Fibonacci sequence the sum of two successive terms gives the third term. Following are the first few terms of the Fibonacci sequence:

1 1 2 3 5 8 13 21 34 55 89...

(d) A positive integer is entered through the keyboard, write a function to find the binary equivalent of this number using recursion.

(e) Write a recursive function to obtain the running sum of first 25 natural numbers.

(f) Write a C function to evaluate the series

$$\sin(x) = x - (x^3 / 3!) + (x^5 / 5!) - (x^7 / 7!) + \Lambda$$

to five significant digits.

(g) Given three variables **x**, **y**, **z** write a function to circularly shift their values to right. In other words if x = 5, y = 8, z = 10 after circular shift y = 5, z = 8, x =10 after circular shift y = 5, z = 8 and x = 10. Call the function with variables **a**, **b**, **c** to circularly shift values.

(h) Write a function to find the binary equivalent of a given decimal integer and display it.

(i) If the lengths of the sides of a triangle are denoted by **a**, **b**, and **c**, then area of triangle is given by

$$area = \sqrt{S(S-a)(S-b)(S-c)}$$

where, S = (a + b + c) / 2

(j) Write a function to compute the distance between two points and use it to develop another function that will compute the area of the triangle whose vertices are **A(x1, y1)**, **B(x2, y2)**, and **C(x3, y3)**. Use these functions to develop a function which returns a value 1 if the point **(x, y)** lines inside the triangle ABC, otherwise a value 0.

(k) Write a function to compute the greatest common divisor given by Euclid's algorithm, exemplified for J = 1980, K = 1617 as follows:

1980 / 1617 = 1	1980 – 1 * 1617 = 363
1617 / 363 = 4	1617 – 4 * 363 = 165
363 / 165 = 2	363 – 2 * 165 = 33
5 / 33 = 5	165 – 5 * 33 = 0

Thus, the greatest common divisor is 33.

6 Data Types Revisited

- Integers, *long* and *short*
- Integers, *signed* and *unsigned*
- Chars, *signed* and *unsigned*
- Floats and Doubles
- A Few More Issues…
- Storage Classes in C
    - Automatic Storage Class
    - Register Storage Class
    - Static Storage Class
    - External Storage Class
    - Which to Use When
- Summary
- Exercise

As seen in the first chapter the primary data types could be of three varieties—**char**, **int**, and **float**. It may seem odd to many, how C programmers manage with such a tiny set of data types. Fact is, the C programmers aren't really deprived. They can derive many data types from these three types. In fact, the number of data types that can be derived in C, is in principle, unlimited. A C programmer can always invent whatever data type he needs.

Not only this, the primary data types themselves could be of several types. For example, a **char** could be an **unsigned char** or a **signed char**. Or an **int** could be a **short int** or a **long int**. Sufficiently confusing? Well, let us take a closer look at these variations of primary data types in this chapter.

To fully define a variable one needs to mention not only its type but also its storage class. In this chapter we would be exploring the different storage classes and their relevance in C programming.

Integers, *long* and *short*

We had seen earlier that the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767. For a 32-bit compiler the range would be –2147483648 to +2147483647. Here a 16-bit compiler means that when it compiles a C program it generates machine language code that is targeted towards working on a 16-bit microprocessor like Intel 8086/8088. As against this, a 32-bit compiler like VC++ generates machine language code that is targeted towards a 32-bit microprocessor like Intel Pentium. Note that this does not mean that a program compiled using Turbo C would not work on 32-bit processor. It would run successfully but at that time the 32-bit processor would work as if it were a 16-bit processor. This happens because a 32-bit processor provides support for programs compiled using 16-bit compilers. If this backward compatibility support is not provided the 16-bit program

would not run on it. This is precisely what happens on the new Intel Itanium processors, which have withdrawn support for 16-bit code.

Remember that out of the two/four bytes used to store an integer, the highest bit ($16^{th}/32^{nd}$ bit) is used to store the sign of the integer. This bit is 1 if the number is negative, and 0 if the number is positive.

C offers a variation of the integer data type that provides what are called **short** and **long** integer values. The intention of providing these variations is to provide integers with different ranges wherever possible. Though not a rule, **short** and **long** integers would usually occupy two and four bytes respectively. Each compiler can decide appropriate sizes depending on the operating system and hardware for which it is being written, subject to the following rules:

(a) **shorts** are at least 2 bytes big
(b) **longs** are at least 4 bytes big
(c) **shorts** are never bigger than **ints**
(d) **ints** are never bigger than **longs**

Figure 6.1 shows the sizes of different integers based upon the OS used.

Compiler	**short**	**int**	**long**
16-bit (Turbo C/C++)	2	2	4
32-bit (Visual C++)	2	4	4

Figure 6.1

long variables which hold **long** integers are declared using the keyword **long**, as in,

```
long int i ;
long int abc ;
```

long integers cause the program to run a bit slower, but the range of values that we can use is expanded tremendously. The value of a **long** integer typically can vary from -2147483648 to +2147483647. More than this you should not need unless you are taking a world census.

If there are such things as **longs**, symmetry requires **shorts** as well—integers that need less space in memory and thus help speed up program execution. **short** integer variables are declared as,

```
short int j ;
short int height ;
```

C allows the abbreviation of **short int** to **short** and of **long int** to **long**. So the declarations made above can be written as,

```
long i ;
long abc ;
short j ;
short height ;
```

Naturally, most C programmers prefer this short-cut.

Sometimes we come across situations where the constant is small enough to be an **int**, but still we want to give it as much storage as a **long**. In such cases we add the suffix 'L' or 'l' at the end of the number, as in 23L.

Integers, *signed* and *unsigned*

Sometimes, we know in advance that the value stored in a given integer variable will always be positive—when it is being used to

only count things, for example. In such a case we can declare the variable to be **unsigned**, as in,

```
unsigned int num_students ;
```

With such a declaration, the range of permissible integer values (for a 16-bit OS) will shift from the range -32768 to +32767 to the range 0 to 65535. Thus, declaring an integer as **unsigned** almost doubles the size of the largest possible value that it can otherwise take. This so happens because on declaring the integer as **unsigned**, the left-most bit is now free and is not used to store the sign of the number. Note that an **unsigned** integer still occupies two bytes. This is how an **unsigned** integer can be declared:

```
unsigned int i ;
unsigned i ;
```

Like an **unsigned int**, there also exists a **short unsigned int** and a **long unsigned int**. By default a **short int** is a **signed short int** and a **long int** is a **signed long int**.

Chars, *signed* and *unsigned*

Parallel to **signed** and **unsigned ints** (either **short** or **long**), similarly there also exist **signed** and **unsigned chars**, both occupying one byte each, but having different ranges. To begin with it might appear strange as to how a **char** can have a sign. Consider the statement

```
char ch = 'A' ;
```

Here what gets stored in **ch** is the binary equivalent of the ASCII value of 'A' (i.e. binary of 65). And if 65's binary can be stored, then -54's binary can also be stored (in a **signed char**).

A **signed char** is same as an ordinary **char** and has a range from -128 to +127; whereas, an **unsigned char** has a range from 0 to 255. Let us now see a program that illustrates this range:

```
main( )
{
    char ch = 291 ;
    printf ( "\n%d %c", ch, ch ) ;
}
```

What output do you expect from this program? Possibly, 291 and the character corresponding to it. Well, not really. Surprised? The reason is that **ch** has been defined as a **char**, and a **char** cannot take a value bigger than +127. Hence when value of **ch** exceeds +127, an appropriate value from the other side of the range is picked up and stored in **ch**. This value in our case happens to be 35, hence 35 and its corresponding character #, gets printed out.

Here is another program that would make the concept clearer.

```
main( )
{
    char ch ;

    for ( ch = 0 ; ch <= 255 ; ch++ )
        printf ( "\n%d %c", ch, ch ) ;
}
```

This program should output ASCII values and their corresponding characters. Well, No! This is an indefinite loop. The reason is that **ch** has been defined as a **char**, and a **char** cannot take values bigger than +127. Hence when value of **ch** is +127 and we perform **ch++** it becomes -128 instead of +128. -128 is less than 255 hence the condition is still satisfied. Here onwards **ch** would take values like -127, -126, -125, ... -2, -1, 0, +1, +2, ... +127, -128, -127, etc. Thus the value of **ch** would keep oscillating between -128 to +127, thereby ensuring that the loop never gets terminated. How do you

overcome this difficulty? Would declaring **ch** as an **unsigned char** solve the problem? Even this would not serve the purpose since when **ch** reaches a value 255, **ch++** would try to make it 256 which cannot be stored in an **unsigned char**. Thus the only alternative is to declare **ch** as an **int**. However, if we are bent upon writing the program using **unsigned char**, it can be done as shown below. The program is definitely less elegant, but workable all the same.

```
main( )
{
    unsigned char  ch ;

    for ( ch = 0 ; ch <= 254 ; ch++ )
        printf ( "\n%d %c", ch, ch ) ;

    printf ( "\n%d %c", ch, ch ) ;
}
```

Floats and Doubles

A **float** occupies four bytes in memory and can range from -3.4e38 to +3.4e38. If this is insufficient then C offers a **double** data type that occupies 8 bytes in memory and has a range from -1.7e308 to +1.7e308. A variable of type **double** can be declared as,

```
double  a, population ;
```

If the situation demands usage of real numbers that lie even beyond the range offered by **double** data type, then there exists a **long double** that can range from -1.7e4932 to +1.7e4932. A **long double** occupies 10 bytes in memory.

You would see that most of the times in C programming one is required to use either **chars** or **ints** and cases where **floats**, **doubles** or **long doubles** would be used are indeed rare.

Let us now write a program that puts to use all the data types that we have learnt in this chapter. Go through the following program carefully, which shows how to use these different data types. Note the format specifiers used to input and output these data types.

```
main( )
{
    char c ;
    unsigned char d ;
    int i ;
    unsigned int j ;
    short int k ;
    unsigned short int l ;
    long int m ;
    unsigned long int n ;
    float x ;
    double y ;
    long double z ;

    /* char */
    scanf ( "%c %c", &c, &d ) ;
    printf ( "%c %c", c, d ) ;

    /* int */
    scanf ( "%d %u", &i, &j ) ;
    printf ( "%d %u", i, j ) ;

    /* short int */
    scanf ( "%d %u", &k, &l ) ;
    printf ( "%d %u", k, l ) ;

    /* long int */
    scanf ( "%ld %lu", &m, &n ) ;
    printf ( "%ld %lu", m, n ) ;

    /* float, double, long double */
    scanf ( "%f %lf %Lf", &x, &y, &z ) ;
    printf ( "%f %lf %Lf", x, y, z ) ;
```

}

The essence of all the data types that we have learnt so far has been captured in Figure 6.2.

Data Type	Range	Bytes	Format
signed char	-128 to + 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf
Note: The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.			

Figure 6.2

A Few More Issues…

Having seen all the variations of the primary types let us take a look at some more related issues.

(a) We saw earlier that size of an integer is compiler dependent. This is even true in case of **chars** and **floats**. Also, depending upon the microprocessor for which the compiler targets its code the accuracy of floating point calculations may change. For example, the result of 22.0/7.0 would be reported more

accurately by VC++ compiler as compared to TC/TC++ compilers. This is because TC/TC++ targets its compiled code to 8088/8086 (16-bit) microprocessors. Since these microprocessors do not offer floating point support, TC/TC++ performs all float operations using a software piece called Floating Point Emulator. This emulator has limitations and hence produces less accurate results. Also, this emulator becomes part of the EXE file, thereby increasing its size. In addition to this increased size there is a performance penalty since this bigger code would take more time to execute.

(b) If you look at ranges of **chars** and **ints** there seems to be one extra number on the negative side. This is because a negative number is always stored as 2's compliment of its binary. For example, let us see how -128 is stored. Firstly, binary of 128 is calculated (10000000), then its 1's compliment is obtained (01111111). A 1's compliment is obtained by changing all 0s to 1s and 1s to 0s. Finally, 2's compliment of this number, i.e. 10000000, gets stored. A 2's compliment is obtained by adding 1 to the 1's compliment. Thus, for -128, 10000000 gets stored. This is an 8-bit number and it can be easily accommodated in a **char**. As against this, +128 cannot be stored in a **char** because its binary 010000000 (left-most 0 is for positive sign) is a 9-bit number. However +127 can be stored as its binary 01111111 turns out to be a 8-bit number.

(c) What happens when we attempt to store +128 in a **char**? The first number on the negative side, i.e. -128 gets stored. This is because from the 9-bit binary of +128, 010000000, only the right-most 8 bits get stored. But when 10000000 is stored the left-most bit is 1 and it is treated as a sign bit. Thus the value of the number becomes -128 since it is indeed the binary of -128, as can be understood from (b) above. Similarly, you can verify that an attempt to store +129 in a **char** results in storing -127 in it. In general, if we exceed the range from positive side we end up on the negative side. Vice versa is

also true. If we exceed the range from negative side we end up on positive side.

Storage Classes in C

We have already said all that needs to be said about constants, but we are not finished with variables. To fully define a variable one needs to mention not only its 'type' but also its 'storage class'. In other words, not only do all variables have a data type, they also have a 'storage class'.

We have not mentioned storage classes yet, though we have written several programs in C. We were able to get away with this because storage classes have defaults. If we don't specify the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used. Thus, variables have certain default storage classes.

From C compiler's point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept— Memory and CPU registers. It is the variable's storage class that determines in which of these two locations the value is stored.

Moreover, a variable's storage class tells us:

(a) Where the variable would be stored.
(b) What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. the default initial value).
(c) What is the scope of the variable; i.e. in which functions the value of the variable would be available.
(d) What is the life of the variable; i.e. how long would the variable exist.

There are four storage classes in C:

(a) Automatic storage class
(b) Register storage class
(c) Static storage class
(d) External storage class

Let us examine these storage classes one by one.

Automatic Storage Class

The features of a variable defined to have an automatic storage class are as under:

Storage	–	Memory.
Default initial value	–	An unpredictable value, which is often called a garbage value.
Scope	–	Local to the block in which the variable is defined.
Life	–	Till the control remains within the block in which the variable is defined.

Following program shows how an automatic storage class variable is declared, and the fact that if the variable is not initialized it contains a garbage value.

```
main( )
{
    auto int i, j ;
    printf ( "\n%d  %d", i, j ) ;
}
```

The output of the above program could be...

```
1211 221
```

where, 1211 and 221 are garbage values of **i** and **j**. When you run this program you may get different values, since garbage values

are unpredictable. So always make it a point that you initialize the automatic variables properly, otherwise you are likely to get unexpected results. Note that the keyword for this storage class is **auto**, and not automatic.

Scope and life of an automatic variable is illustrated in the following program.

```
main( )
{
    auto int  i = 1 ;
    {
        {
            {
                printf ( "\n%d ", i ) ;
            }
            printf ( "%d ", i ) ;
        }
        printf ( "%d", i ) ;
    }
}
```

The output of the above program is:

1 1 1

This is because, all **printf( )** statements occur within the outermost block (a block is all statements enclosed within a pair of braces) in which **i** has been defined. It means the scope of **i** is local to the block in which it is defined. The moment the control comes out of the block in which the variable is defined, the variable and its value is irretrievably lost. To catch my point, go through the following program.

```
main( )
{
    auto int  i = 1 ;
    {
```

```
        auto int i = 2 ;
        {
            auto int i = 3 ;
            printf ( "\n%d ", i ) ;
        }
        printf ( "%d ", i ) ;
    }
    printf ( "%d", i ) ;
}
```

The output of the above program would be:

```
3 2 1
```

Note that the Compiler treats the three **i**'s as totally different variables, since they are defined in different blocks. Once the control comes out of the innermost block the variable **i** with value 3 is lost, and hence the **i** in the second **printf( )** refers to **i** with value 2. Similarly, when the control comes out of the next innermost block, the third **printf( )** refers to the **i** with value 1.

Understand the concept of life and scope of an automatic storage class variable thoroughly before proceeding with the next storage class.

Register Storage Class

The features of a variable defined to be of **register** storage class are as under:

Storage	-	CPU registers.
Default initial value	-	Garbage value.
Scope	-	Local to the block in which the variable is defined.
Life	-	Till the control remains within the block in which the variable is defined.

A value stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as **register**. A good example of frequently used variables is loop counters. We can name their storage class as **register**.

```
main( )
{
    register int i ;

    for ( i = 1 ; i <= 10 ; i++ )
        printf ( "\n%d", i ) ;
}
```

Here, even though we have declared the storage class of **i** as **register**, we cannot say for sure that the value of **i** would be stored in a CPU register. Why? Because the number of CPU registers are limited, and they may be busy doing some other task. What happens in such an event... the variable works as if its storage class is **auto**.

Not every type of variable can be stored in a CPU register.

For example, if the microprocessor has 16-bit registers then they cannot hold a **float** value or a **double** value, which require 4 and 8 bytes respectively. However, if you use the **register** storage class for a **float** or a **double** variable you won't get any error messages. All that would happen is the compiler would treat the variables to be of **auto** storage class.

Static Storage Class

The features of a variable defined to have a **static** storage class are as under:

Storage – Memory.
Default initial value – Zero.

Scope – Local to the block in which the variable is defined.

Life – Value of the variable persists between different function calls.

Compare the two programs and their output given in Figure 6.3 to understand the difference between the **automatic** and **static** storage classes.

```
main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
}

increment( )
{
    auto int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

```
main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
}

increment( )
{
    static int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

The output of the above programs would be:

```
1
1
1
```

```
1
2
3
```

Figure 6.3

The programs above consist of two functions **main( )** and **increment( )**. The function **increment( )** gets called from **main( )** thrice. Each time it increments the value of **i** and prints it. The only difference in the two programs is that one uses an **auto** storage class for variable **i**, whereas the other uses **static** storage class.

Like **auto** variables, **static** variables are also local to the block in which they are declared. The difference between them is that **static** variables don't disappear when the function is no longer active. Their values persist. If the control comes back to the same function again the **static** variables have the same values they had last time around.

In the above example, when variable **i** is **auto**, each time **increment( )** is called it is re-initialized to one. When the function terminates, **i** vanishes and its new value of 2 is lost. The result: no matter how many times we call **increment( )**, **i** is initialized to 1 every time.

On the other hand, if **i** is **static**, it is initialized to 1 only once. It is never initialized again. During the first call to **increment( )**, **i** is incremented to 2. Because **i** is static, this value persists. The next time **increment( )** is called, **i** is not re-initialized to 1; on the contrary its old value 2 is still available. This current value of **i** (i.e. 2) gets printed and then **i = i + 1** adds 1 to **i** to get a value of 3. When **increment( )** is called the third time, the current value of **i** (i.e. 3) gets printed and once again **i** is incremented. In short, if the storage class is **static** then the statement **static int i = 1** is executed only once, irrespective of how many times the same function is called.

Consider one more program.

```
main( )
{
    int *j ;
    int * fun( ) ;
    j = fun( ) ;
    printf ( "\n%d", *j ) ;
}

int *fun( )
{
```

```
    int k = 35 ;
    return ( &k ) ;
}
```

Here we are returning an address of **k** from **fun( )** and collecting it in **j**. Thus **j** becomes pointer to **k**. Then using this pointer we are printing the value of **k**. This correctly prints out 35. Now try calling any function (even **printf( )**) immediately after the call to **fun( )**. This time **printf( )** prints a garbage value. Why does this happen? In the first case, when the control returned from **fun( )** though **k** went dead it was still left on the stack. We then accessed this value using its address that was collected in **j**. But when we precede the call to **printf( )** by a call to any other function, the stack is now changed, hence we get the garbage value. If we want to get the correct value each time then we must declare **k** as **static**. By doing this when the control returns from **fun( )**, **k** would not die.

All this having been said, a word of advice—avoid using **static** variables unless you really need them. Because their values are kept in memory when the variables are not active, which means they take up space in memory that could otherwise be used by other variables.

External Storage Class

The features of a variable whose storage class has been defined as external are as follows:

Storage	–	Memory.
Default initial value	–	Zero.
Scope	–	Global.
Life	–	As long as the program's execution doesn't come to an end.

External variables differ from those we have already discussed in that their scope is global, not local. External variables are declared outside all functions, yet are available to all functions that care to use them. Here is an example to illustrate this fact.

```
int i ;
main( )
{
    printf ( "\ni = %d", i ) ;

    increment( ) ;
    increment( ) ;
    decrement( ) ;
    decrement( ) ;
}

increment( )
{
    i = i + 1 ;
    printf ( "\non incrementing i = %d", i ) ;
}

decrement( )
{
    i = i - 1 ;
    printf ( "\non decrementing i = %d", i ) ;
}
```

The output would be:

```
i = 0
on incrementing i = 1
on incrementing i = 2
on decrementing i = 1
on decrementing i = 0
```

As is obvious from the above output, the value of **i** is available to the functions **incremeht( )** and **decrement( )** since **i** has been declared outside all functions.

Look at the following program.

```
int x = 21 ;
main( )
{
    extern int y ;
    printf ( "\n%d %d", x, y ) ;
}
int y = 31 ;
```

Here, **x** and **y** both are global variables. Since both of them have been defined outside all the functions both enjoy external storage class. Note the difference between the following:

```
extern int y ;
int y = 31 ;
```

Here the first statement is a declaration, whereas the second is the definition. When we declare a variable no space is reserved for it, whereas, when we define it space gets reserved for it in memory. We had to declare **y** since it is being used in **printf( )** before it's definition is encountered. There was no need to declare **x** since its definition is done before its usage. Also remember that a variable can be declared several times but can be defined only once.

Another small issue—what will be the output of the following program?

```
int x = 10 ;
main( )
{
    int x = 20 ;

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

```
    display( ) ;
}
display( )
{
    printf ( "\n%d", x ) ;
}
```

Here **x** is defined at two places, once outside **main( )** and once inside it. When the control reaches the **printf( )** in **main( )** which **x** gets printed? Whenever such a conflict arises, it's the local variable that gets preference over the global variable. Hence the **printf( )** outputs 20. When **display( )** is called and control reaches the **printf( )** there is no such conflict. Hence this time the value of the global **x**, i.e. 10 gets printed.

One last thing—a **static** variable can also be declared outside all the functions. For all practical purposes it will be treated as an **extern** variable. However, the scope of this variable is limited to the same file in which it is declared. This means that the variable would not be available to any function that is defined in a file other than the file in which the variable is defined.

Which to Use When

Dennis Ritchie has made available to the C programmer a number of storage classes with varying features, believing that the programmer is in a best position to decide which one of these storage classes is to be used when. We can make a few ground rules for usage of different storage classes in different programming situations with a view to:

(a) economise the memory space consumed by the variables
(b) improve the speed of execution of the program

The rules are as under:

- Use **static** storage class only if you want the value of a variable to persist between different function calls.
- Use **register** storage class for only those variables that are being used very often in a program. Reason is, there are very few CPU registers at our disposal and many of them might be busy doing something else. Make careful utilization of the scarce resources. A typical application of **register** storage class is loop counters, which get used a number of times in a program.
- Use **extern** storage class for only those variables that are being used by almost all the functions in the program. This would avoid unnecessary passing of these variables as arguments when making a function call. Declaring all the variables as **extern** would amount to a lot of wastage of memory space because these variables would remain active throughout the life of the program.
- If you don't have any of the express needs mentioned above, then use the **auto** storage class. In fact most of the times we end up using the **auto** variables, because often it so happens that once we have used the variables in a function we don't mind loosing them.

Summary

(a) We can use different variations of the primary data types, namely **signed** and **unsigned char**, **long** and **short int**, **float**, **double** and **long double**. There are different format specifications for all these data types when they are used in **scanf( )** and **printf( )** functions.

(b) The maximum value a variable can hold depends upon the number of bytes it occupies in memory.

(c) By default all the variables are **signed**. We can declare a variable as **unsigned** to accommodate greater value without increasing the bytes occupied.

(d) We can make use of proper storage classes like **auto**, **register**, **static** and **extern** to control four properties of the variable—storage, default initial value, scope and life.

Exercise

[A] What would be the output of the following programs:

(a)
```
main( )
{
    int i ;
    for ( i = 0 ; i <= 50000 ; i++ )
        printf ( "\n%d", i ) ;
}
```

(b)
```
main( )
{
    float  a = 13.5 ;
    double  b = 13.5 ;
    printf ( "\n%f %lf", a, b ) ;
}
```

(c)
```
int  i = 0 ;
main( )
{
    printf ( "\nmain's i = %d", i ) ;
    i++ ;
    val( ) ;
    printf ( "\nmain's i = %d", i ) ;
    val( ) ;
}
val( )
{
    i = 100 ;
    printf ( "\nval's i = %d", i ) ;
    i++ ;
}
```

(d)
```
main( )
{
    int x, y, s = 2 ;
    s *= 3 ;
    y = f ( s ) ;
    x = g ( s ) ;
    printf ( "\n%d %d %d", s, y, x ) ;
}
int t = 8 ;
f ( int a )
{
    a += -5 ;
    t -= 4 ;
    return ( a + t ) ;
}
g ( int a )
{
    a = 1 ;
    t += a ;
    return ( a + t ) ;
}
```

(e)
```
main( )
{
    static int count = 5 ;
    printf ( "\ncount = %d", count-- ) ;
    if ( count != 0 )
        main( ) ;
}
```

(f)
```
main( )
{
    int i, j ;
    for ( i = 1 ; i < 5 ; i++ )
    {
        j = g ( i ) ;
        printf ( "\n%d", j ) ;
    }
```

```
}
g ( int x )
{
    static int v = 1 ;
    int b = 3 ;
    v += x ;
    return ( v + x + b ) ;
}
```

(g)
```
float x = 4.5 ;
main( )
{
    float y, float f ( float ) ;
    x *= 2.0 ;
    y = f ( x ) ;
    printf ( "\n%f %f", x, y ) ;
}
float f ( float a )
{
    a += 1.3 ;
    x -= 4.5 ;
    return ( a + x ) ;
}
```

(h)
```
main( )
{
    func( ) ;
    func( ) ;
}
func( )
{
    auto int i = 0 ;
    register int j = 0 ;
    static int k = 0 ;
    i++ ; j++ ; k++ ;
    printf ( "\n %d % d %d", i, j, k ) ;
}
```

(i)
```
int x = 10 ;
main( )
{
    int x = 20 ;
    {
        int x  = 30 ;
        printf ( "\n%d", x ) ;
    }
    printf ("\n%d", x ) ;
}
```

[B] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    long  num ;
    num = 2 ;
    printf ( "\n%ld", num ) ;
}
```

(b)
```
main( )
{
    char  ch = 200 ;
    printf ( "\n%d", ch ) ;
}
```

(c)
```
main( )
{
    unsigned  a = 25 ;
    long unsigned  b = 25l ;
    printf ( "\n%lu %u", a, b ) ;
}
```

(d)
```
main( )
{
    long float  a = 25.345e454 ;
    unsigned double  b = 25 ;
    printf ( "\n%lf %d", a, b ) ;
```

```
    }
```

(e)
```
main( )
{
    float a = 25.345 ;
    float *b ;
    b = &a ;
    printf ( "\n%f %u", a, b ) ;
}
```

(f)
```
static int y ;
main( )
{
    static int z ;
    printf ("%d %d", y, z ) ;
}
```

[C] State whether the following statements are True or False:

(a) Storage for a register storage class variable is allocated each time the control reaches the block in which it is present.

(b) An extern storage class variable is not available to the functions that precede its definition, unless the variable is explicitly declared in these functions.

(c) The value of an automatic storage class variable persists between various function invocations.

(d) If the CPU registers are not available, the register storage class variables are treated as static storage class variables.

(e) The register storage class variables cannot hold float values.

(f) If we try to use register storage class for a **float** variable the compiler will flash an error message.

(g) If the variable **x** is defined as **extern** and a variable **x** is also defined as a local variable of some function, then the global variable gets preference over the local variable.

(h) The default value for automatic variable is zero.

(i) The life of static variable is till the control remains within the block in which it is defined.

(j) If a global variable is to be defined, then the **extern** keyword is necessary in its declaration.

(k) The address of register variable is not accessible.

[D] Following program calculates the sum of digits of the number 12345. Go through it and find out why is it necessary to declare the storage class of the variable **sum** as **static**.

```
main( )
{
    int a ;
    a = sumdig ( 12345 ) ;
    printf ( "\n%d", a ) ;
}
sumdig ( int num )
{
    static int sum ;
    int a, b ;
    a = num % 10 ;
    b = ( num - a ) / 10 ;
    sum = sum + a ;
    if ( b != 0 )
        sumdig ( b ) ;
    else
        return ( sum ) ;
}
```

7 The C Preprocessor

- Features of C Preprocessor
- Macro Expansion
    - Macros with Arguments
    - Macros versus Functions
- File Inclusion
- Conditional Compilation
- *#if* and *#elif* Directives
- Miscellaneous Directives
    - *#undef* Directive
    - *#pragma* Directive
- Summary
- Exercise

The C preprocessor is exactly what its name implies. It is a program that processes our source program before it is passed to the compiler. Preprocessor commands (often known as directives) form what can almost be considered a language within C language. We can certainly write C programs without knowing anything about the preprocessor or its facilities. But preprocessor is such a great convenience that virtually all C programmers rely on it. This chapter explores the preprocessor directives and discusses the pros and cons of using them in programs.

Features of C Preprocessor

There are several steps involved from the stage of writing a C program to the stage of getting it executed. Figure 7.1 shows these different steps along with the files created during each stage. You can observe from the figure that our program passes through several processors before it is ready to be executed. The input and output to each of these processors is shown in Figure 7.2.

Note that if the source code is stored in a file PR1.C then the expanded source code gets stored in a file PR1.I. When this expanded source code is compiled the object code gets stored in PR1.OBJ. When this object code is linked with the object code of library functions the resultant executable code gets stored in PR1.EXE.

The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition. We would learn the following preprocessor directives here:

(a) Macro expansion
(b) File inclusion

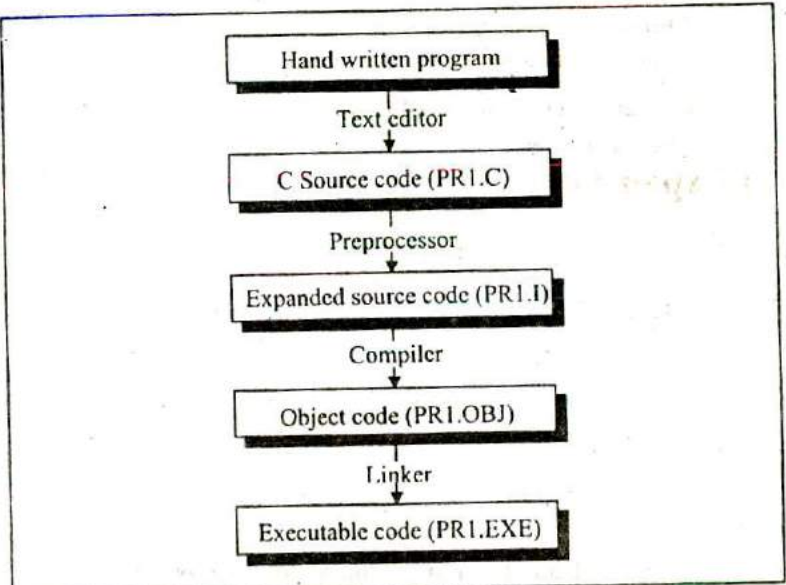


Figure 7.1

Processor	Input	Output
Editor	Program typed from keyboard	C source code containing program and preprocessor commands
Prepro-cessor	C source code file	Source code file with the preprocessing commands properly sorted out
Compiler	Source code file with preprocessing commands sorted out	Relocatable object code
Linker	Relocatable object code and the standard C library functions	Executable code in machine language

Figure 7.2

(c) Conditional Compilation
(d) Miscellaneous directives

Let us understand these features of preprocessor one by one.

Macro Expansion

Have a look at the following program.

```
#define UPPER 25
main( )
{
    int i ;
    for ( i = 1 ; i <= UPPER ; i++ )
        printf ( "\n%d", i ) ;
}
```

In this program instead of writing 25 in the **for** loop we are writing it in the form of UPPER, which has already been defined before **main( )** through the statement,

```
#define UPPER 25
```

This statement is called 'macro definition' or more commonly, just a 'macro'. What purpose does it serve? During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25. Here is another example of macro definition.

```
#define PI 3.1415
main( )
{
    float r = 6.25 ;
    float area ;

    area = PI * r * r ;
    printf ( "\nArea of circle = %f", area ) ;
}
```

UPPER and PI in the above programs are often called 'macro templates', whereas, 25 and 3.1415 are called their corresponding 'macro expansions'.

When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the **#define** directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

In C programming it is customary to use capital letters for macro template. This makes it easy for programmers to pick out all the macro templates when reading through the program.

Note that a macro template and its macro expansion are separated by blanks or tabs. A space between # and **define** is optional. Remember that a macro definition is never to be terminated by a semicolon.

And now a million dollar question... why use **#define** in the above programs? What have we gained by substituting PI for 3.1415 in our program? Probably, we have made the program easier to read. Even though 3.1415 is such a common constant that it is easily recognizable, there are many instances where a constant doesn't reveal its purpose so readily. For example, if the phrase "\x1B[2J" causes the screen to clear. But which would you find easier to understand in the middle of your program "\x1B[2J" or "CLEARSCREEN"? Thus, we would use the macro definition

```
#define CLEARSCREEN "\x1B[2J"
```

Then wherever CLEARSCREEN appears in the program it would automatically be replaced by "\x1B[2J" before compilation begins.

There is perhaps a more important reason for using macro definition than mere readability. Suppose a constant like 3.1415 appears many times in your program. This value may have to be changed some day to 3.141592. Ordinarily, you would need to go through the program and manually change each occurrence of the constant. However, if you have defined PI in a **#define** directive, you only need to make one change, in the **#define** directive itself:

```
#define PI 3.141592
```

Beyond this the change will be made automatically to all occurrences of PI before the beginning of compilation.

In short, it is nice to know that you would be able to change values of a constant at all the places in the program by just making a change in the **#define** directive. This convenience may not matter for small programs shown above, but with large programs macro definitions are almost indispensable.

But the same purpose could have been served had we used a variable **pi** instead of a macro template **PI**. A variable could also have provided a meaningful name for a constant and permitted one change to effect many occurrences of the constant. It's true that a variable can be used in this way. Then, why not use it? For three reasons it's a bad idea.

Firstly, it is inefficient, since the compiler can generate faster and more compact code for constants than it can for variables. Secondly, using a variable for what is really a constant encourages sloppy thinking and makes the program more difficult to understand: if something never changes, it is hard to imagine it as a variable. And thirdly, there is always a danger that the variable may inadvertently get altered somewhere in the program. So it's no longer a constant that you think it is.

Thus, using **#define** can produce more efficient and more easily understandable programs. This directive is used extensively by C programmers, as you will see in many programs in this book.

Following three examples show places where a **#define** directive is popularly used by C programmers.

A **#define** directive is many a times used to define operators as shown below.

```
#define AND &&
#define OR ||
main( )
{
    int f = 1, x = 4, y = 90 ;

    if ( ( f < 5 ) AND ( x <= 20 OR y <= 45 ) )
        printf ( "\nYour PC will always work fine..." ) ;
    else
        printf ( "\nIn front of the maintenance man" ) ;
}
```

A **#define** directive could be used even to replace a condition, as shown below.

```
#define AND &&
#define ARANGE ( a > 25 AND a < 50 )
main( )
{
    int a = 30 ;

    if ( ARANGE )
        printf ( "within range" ) ;
    else
        printf ( "out of range" ) ;
}
```

A **#define** directive could be used to replace even an entire C statement. This is shown below.

```
#define FOUND printf ( "The Yankee Doodle Virus" ) ;
main( )
{
    char signature ;

    if ( signature == 'Y' )
        FOUND
    else
        printf ( "Safe... as yet !" ) ;
}
```

Macros with Arguments

The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can. Here is an example that illustrates this fact.

```
#define AREA(x) ( 3.14 * x * x )
main( )
{
    float r1 = 6.25, r2 = 2.5, a ;

    a = AREA ( r1 ) ;
    printf ( "\nArea of circle = %f", a ) ;
    a = AREA ( r2 ) ;
    printf ( "\nArea of circle = %f", a ) ;
}
```

Here's the output of the program...

```
Area of circle = 122.656250
Area of circle = 19.625000
```

In this program wherever the preprocessor finds the phrase **AREA(x)** it expands it into the statement **(3.14 * x * x)**. However, that's not all that it does. The **x** in the macro template **AREA(x)** is an argument that matches the **x** in the macro expansion **(3.14 * x * x)**. The statement **AREA(r1)** in the program causes the variable **r1** to be substituted for **x**. Thus the statement **AREA(r1)** is equivalent to:

```
( 3.14 * r1 * r1 )
```

After the above source code has passed through the preprocessor, what the compiler gets to work on will be this:

```
main( )
{
    float r1 = 6.25, r2 = 2.5, a ;

    a = 3.14 * r1 *r1 ;
    printf ( "Area of circle = %f\n", a ) ;
    a = 3.14 *r2 * r2 ;
    printf ( "Area of circle = %f", a ) ;
}
```

Here is another example of macros with arguments:

```
#define ISDIGIT(y) ( y >= 48 && y <= 57 )
main( )
{
    char ch ;

    printf ( "Enter any digit " ) ;
    scanf ( "%c", &ch ) ;

    if ( ISDIGIT ( ch ) )
        printf ( "\nYou entered a digit" ) ;
    else
        printf ( "\nIllegal input" ) ;
```

```
}
```

Here are some important points to remember while writing macros with arguments:

(a) Be careful not to leave a blank between the macro template and its argument while defining the macro. For example, there should be no blank between **AREA** and **(x)** in the definition, #define AREA(x) (3.14 * x * x)

If we were to write **AREA (x)** instead of **AREA(x)**, the **(x)** would become a part of macro expansion, which we certainly don't want. What would happen is, the template would be expanded to

```
( r1 ) ( 3.14 * r1 * r1 )
```

which won't run. Not at all what we wanted.

(b) The entire macro expansion should be enclosed within parentheses. Here is an example of what would happen if we fail to enclose the macro expansion within parentheses.

```
#define SQUARE(n) n * n
main( )
{
    int j ;

    j = 64 / SQUARE ( 4 ) ;
    printf ( "j = %d", j ) ;
}
```

The output of the above program would be:

```
j = 64
```

whereas, what we expected was j = 4.

What went wrong? The macro was expanded into

```
j = 64 / 4 * 4 ;
```

which yielded 64.

(c) Macros can be split into multiple lines, with a '\' (back slash) present at the end of each line. Following program shows how we can define and use multiple line macros.

```
#define HLINE  for ( i = 0 ; i < 79 ; i++ ) \
                    printf ( "%c", 196 ) ;

#define VLINE( X, Y ) {\
                        gotoxy ( X, Y ) ; \
                        printf ( "%c", 179 ) ; \
                    }
main( )
{
   int i, y ;
   clrscr( ) ;

   gotoxy ( 1, 12 ) ;
   HLINE

   for ( y = 1 ; y < 25 ; y++ )
      VLINE ( 39, y ) ;
}
```

This program draws a vertical and a horizontal line in the center of the screen.

(d) If for any reason you are unable to debug a macro then you should view the expanded code of the program to see how the macros are getting expanded. If your source code is present in the file PR1.C then the expanded source code would be stored

in PR1.I. You need to generate this file at the command prompt by saying:

```
cpp pr1.c
```

Here CPP stands for C PreProcessor. It generates the expanded source code and stores it in a file called PR1.I. You can now open this file and see the expanded source code. Note that the file PR1.I gets generated in C:\TC\BIN directory. The procedure for generating expanded source code for compilers other than Turbo C/C++ might be a little different.

Macros versus Functions

In the above example a macro was used to calculate the area of the circle. As we know, even a function can be written to calculate the area of the circle. Though macro calls are 'like' function calls, they are not really the same things. Then what is the difference between the two?

In a macro call the preprocessor replaces the macro template with its macro expansion, in a stupid, unthinking, literal way. As against this, in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function.

This brings us to a question: when is it best to use macros with arguments and when is it better to use a function? Usually macros make the program run faster but increase the program size, whereas functions make the program smaller and compact.

If we use a macro hundred times in a program, the macro expansion goes into our source code at hundred different places, thus increasing the program size. On the other hand, if a function is used, then even if it is called from hundred different places in

the program, it would take the same amount of space in the program.

But passing arguments to a function and getting back the returned value does take time and would therefore slow down the program. This gets avoided with macros since they have already been expanded and placed in the source code before compilation.

Moral of the story is—if the macro is simple and sweet like in our examples, it makes nice shorthand and avoids the overheads associated with function calls. On the other hand, if we have a fairly large macro and it is used fairly often, perhaps we ought to replace it with a function.

File Inclusion

The second preprocessor directive we'll explore in this chapter is file inclusion. This directive causes one file to be included in another. The preprocessor command for file inclusion looks like this:

```
#include "filename"
```

and it simply causes the entire contents of **filename** to be inserted into the source code at that point in the program. Of course this presumes that the file being included is existing. When and why this feature is used? It can be used in two cases:

(a) If we have a very large program, the code is best divided into several different files, each containing a set of related functions. It is a good programming practice to keep different sections of a large program separate. These files are **#included** at the beginning of main program file.

(b) There are some functions and some macro definitions that we need almost in all programs that we write. These commonly

needed functions and macro definitions can be stored in a file, and that file can be included in every program we write, which would add all the statements in this file to our program as if we have typed them in.

It is common for the files that are to be included to have a .h extension. This extension stands for 'header file', possibly because it contains statements which when included go to the head of your program. The prototypes of all the library functions are grouped into different categories and then stored in different header files. For example prototypes of all mathematics related functions are stored in the header file 'math.h', prototypes of console input/output functions are stored in the header file 'conio.h', and so on.

Actually there exist two ways to write **#include** statement. These are:

```
#include "filename"
#include <filename>
```

The meaning of each of these forms is given below:

#include "goto.c"	This command would look for the file **goto.c** in the current directory as well as the specified list of directories as mentioned in the include search path that might have been set up.
#include <goto.c>	This command would look for the file **goto.c** in the specified list of directories only.

The include search path is nothing but a list of directories that would be searched for the file being included. Different C compilers let you set the search path in different manners. If you are using Turbo C/C++ compiler then the search path can be set up by selecting 'Directories' from the 'Options' menu. On doing this

a dialog box appears. In this dialog box against 'Include Directories' we can specify the search path. We can also specify multiple include paths separated by ';' (semicolon) as shown below:

```
c:\tc\lib ; c:\mylib ; d:\libfiles
```

The path can contain maximum of 127 characters. Both relative and absolute paths are valid. For example '..\dir\incfiles' is a valid path.

Conditional Compilation

We can, if we want, have the compiler skip over part of a source code by inserting the preprocessing commands **#ifdef** and **#endif**, which have the general form:

```
#ifdef macroname
    statement 1 ;
    statement 2 ;
    statement 3 ;
#endif
```

If **macroname** has been **#defined**, the block of code will be processed as usual; otherwise not.

Where would **#ifdef** be useful? When would you like to compile only a part of your program? In three cases:

(a) To "comment out" obsolete lines of code. It often happens that a program is changed at the last minute to satisfy a client. This involves rewriting some part of source code to the client's satisfaction and deleting the old code. But veteran programmers are familiar with the clients who change their mind and want the old code back again just the way it was.

Now you would definitely not like to retype the deleted code again.

One solution in such a situation is to put the old code within a pair of /* */ combination. But we might have already written a comment in the code that we are about to "comment out". This would mean we end up with nested comments. Obviously, this solution won't work since we can't nest comments in C.

Therefore the solution is to use conditional compilation as shown below.

```
main( )
{
   #ifdef OKAY
        statement 1 ;
        statement 2 ; /* detects virus */
        statement 3 ;
        statement 4 ; /* specific to stone virus */
   #endif

   statement 5 ;
   statement 6 ;
   statement 7 ;
}
```

Here, statements 1, 2, 3 and 4 would get compiled only if the macro OKAY has been defined, and we have purposefully omitted the definition of the macro OKAY. At a later date, if we want that these statements should also get compiled all that we are required to do is to delete the **#ifdef** and **#endif** statements.

(b) A more sophisticated use of **#ifdef** has to do with making the programs portable, i.e. to make them work on two totally different computers. Suppose an organization has two

different types of computers and you are expected to write a program that works on both the machines. You can do so by isolating the lines of code that must be different for each machine by marking them off with **#ifdef**. For example:

```
main( )
{
   #ifdef INTEL
        code suitable for a Intel PC
   #else
        code suitable for a Motorola PC
   #endif
   code common to both the computers
}
```

When you compile this program it would compile only the code suitable for a Intel PC and the common code. This is because the macro INTEL has not been defined. Note that the working of **#ifdef - #else - #endif** is similar to the ordinary **if - else** control instruction of C.

If you want to run your program on a Motorola PC, just add a statement at the top saying,

```
#define INTEL
```

Sometimes, instead of **#ifdef** the **#ifndef** directive is used. The **#ifndef** (which means 'if not defined') works exactly opposite to **#ifdef**. The above example if written using **#ifndef**, would look like this:

```
main( )
{
   #ifndef INTEL
        code suitable for a Intel PC
   #else
        code suitable for a Motorola PC
```

```
    #endif
    code common to both the computers
}
```

(c) Suppose a function **myfunc( )** is defined in a file 'myfile.h' which is **#include**d in a file 'myfile1.h'. Now in your program file if you **#include** both 'myfile.h' and 'myfile1.h' the compiler flashes an error 'Multiple declaration for **myfunc**'. This is because the same file 'myfile.h' gets included twice. To avoid this we can write following code in the header file.

```
/* myfile.h */
#ifndef __myfile_h
    #define __myfile_h

    myfunc( )
    {
        /* some code */
    }

#endif
```

First time the file 'myfile.h' gets included the preprocessor checks whether a macro called **__myfile_h** has been defined or not. If it has not been then it gets defined and the rest of the code gets included. Next time we attempt to include the same file, the inclusion is prevented since **__myfile_h** already stands defined. Note that there is nothing special about **__myfile_h**. In its place we can use any other macro as well.

#if and *#elif* Directives

The **#if** directive can be used to test whether an expression evaluates to a nonzero value or not. If the result of the expression is nonzero, then subsequent lines upto a **#else**, **#elif** or **#endif** are compiled, otherwise they are skipped.

A simple example of **#if** directive is shown below:

```
main( )
{
    #if TEST <= 5
        statement 1 ;
        statement 2 ;
        statement 3 ;
    #else
        statement 4 ;
        statement 5 ;
        statement 6 ;
    #endif
}
```

If the expression, **TEST <= 5** evaluates to true then statements 1, 2 and 3 are compiled otherwise statements 4, 5 and 6 are compiled. In place of the expression **TEST <= 5** other expressions like **(LEVEL == HIGH || LEVEL == LOW)** or **ADAPTER == CGA** can also be used.

If we so desire we can have nested conditional compilation directives. An example that uses such directives is shown below.

```
#if ADAPTER == VGA
    code for video graphics array
#else
    #if ADAPTER == SVGA
        code for super video graphics array
    #else
        code for extended graphics adapter
    #endif
#endif
```

The above program segment can be made more compact by using another conditional compilation directive called **#elif**. The same program using this directive can be rewritten as shown below.

Observe that by using the **#elif** directives the number of **#endif**s used in the program get reduced.

```
#if ADAPTER == VGA
    code for video graphics array
#elif ADAPTER == SVGA
    code for super video graphics array
#else
    code for extended graphics adapter
#endif
```

Miscellaneous Directives

There are two more preprocessor directives available, though they are not very commonly used. They are:

(a) #undef
(b) #pragma

#undef Directive

On some occasions it may be desirable to cause a defined name to become 'undefined'. This can be accomplished by means of the **#undef** directive. In order to undefine a macro that has been earlier **#defined**, the directive,

```
#undef macro template
```

can be used. Thus the statement,

```
#undef PENTIUM
```

would cause the definition of PENTIUM to be removed from the system. All subsequent **#ifdef PENTIUM** statements would evaluate to false. In practice seldom are you required to undefine a macro, but for some reason if you are required to, then you know that there is something to fall back upon.

#pragma Directive

This directive is another special-purpose directive that you can use to turn on or off certain features. Pragmas vary from one compiler to another. There are certain pragmas available with Microsoft C compiler that deal with formatting source listings and placing comments in the object file generated by the compiler. Turbo C/C++ compiler has got a pragma that allows you to suppress warnings generated by the compiler. Some of these pragmas are discussed below.

(a) **#pragma startup** and **#pragma exit:** These directives allow us to specify functions that are called upon program startup (before **main( )**) or program exit (just before the program terminates). Their usage is as follows:

```
void fun1( ) ;
void fun2( ) ;

#pragma startup fun1
#pragma exit fun2

main( )
{
    printf ( "\nInside maim" ) ;
}

void fun1( )
{
    printf ( "\nInside fun1" ) ;
}

void fun2( )
{
    printf ( "\nInside fun2" ) ;
}
```

And here is the output of the program.

```
Inside fun1
Inside main
Inside fun2
```

Note that the functions **fun1( )** and **fun2( )** should neither receive nor return any value. If we want two functions to get executed at startup then their pragmas should be defined in the reverse order in which you want to get them called.

(b) **#pragma warn:** This directive tells the compiler whether or not we want to suppress a specific warning. Usage of this pragma is shown below.

```
#pragma warn -rvl   /* return value */
#pragma warn -par   /* parameter not used */
#pragma warn -rch   /* unreachable code */

int f1( )
{
    int a = 5 ;
}

void f2 ( int x )
{
    printf ( "\nInside f2" ) ;
}

int f3( )
{
    int x = 6 ;
    return x ;
    x++ ;
}

void main( )
```

```
{
    f1( ) ;
    f2 ( 7 ) ;
    f3( ) ;
}
```

If you go through the program you can notice three problems immediately. These are:

(a) Though promised, **f1( )** doesn't return a value.

(b) The parameter **x** that is passed to **f2( )** is not being used anywhere in **f2( )**.

(c) The control can never reach **x++** in **f3( )**.

If we compile the program we should expect warnings indicating the above problems. However, this does not happen since we have suppressed the warnings using the **#pragma** directives. If we replace the '–' sign with a '+' then these warnings would be flashed on compilation. Though it is a bad practice to suppress warnings, at times it becomes useful to suppress them. For example, if you have written a huge program and are trying to compile it, then to begin with you are more interested in locating the errors, rather than the warnings. At such times you may suppress the warnings. Once you have located all errors, then you may turn on the warnings and sort them out.

Summary

(a) The preprocessor directives enable the programmer to write programs that are easy to develop, read, modify and transport to a different computer system.

(b) We can make use of various preprocessor directives such as **#define**, **#include**; **#ifdef - #else - #endif**, **#if** and **#elif** in our program.
(c) The directives like **#undef** and **#pragma** are also useful although they are seldom used.

Exercise

[A] Answer the following:

(a) What is a preprocessor directive

1. a message from compiler to the programmer
2. a message from compiler to the linker
3. a message from programmer to the preprocessor
4. a message from programmer to the microprocessor

(b) Which of the following are correctly formed **#define** statements:

```
#define  INCH PER FEET  12
#define  SQR (X)  ( X * X )
#define  SQR(X)  X * X
#define  SQR(X)  ( X * X )
```

c) State True or False:

1. A macro must always be written in capital letters.
2. A macro should always be accomodated in a single line.
3. After preprocessing when the program is sent for compilation the macros are removed from the expanded source code.
4. Macros with arguments are not allowed.
5. Nested macros are allowed.
6. In a macro call the control is passed to the macro.

(d) How many **#include** directives can be there in a given program file?

(e) What is the difference between the following two **#include** directives:

```
#include "conio.h"
#include <conio.h>
```

(f) A header file is:

1. A file that contains standard library functions
2. A file that contains definitions and macros
3. A file that contains user - defined functions
4. A file that is present in current working directory

(g) Which of the following is not a preprocessor directive

1. #if
2. #elseif
3. #undef
4. #pragma

(h) All macro substitutions in a program are done

1. Before compilation of the program
2. After compilation
3. During execution
4. None of the above

(i) In a program the statement:

```
#include "filename"
```

is replaced by the contents of the file "filename"

1. Before compilation
2. After Compilation
3. During execution
4. None of the above

[B] What would be the output of the following program:

(a)
```
main( )
{
    int i = 2 ;
    #ifdef DEF
        i *= i ;
    #else
        printf ( "\n%d", i ) ;
    #endif
}
```

(b)
```
#define PRODUCT(x) ( x * x )
main( )
{
    int i = 3, j ;
    j = PRODUCT( i + 1 ) ;
    printf ( "\n%d", j ) ;
}
```

(c)
```
#define PRODUCT(x) ( x * x )
main( )
{
    int i = 3, j, k ;
    j = PRODUCT( i++ ) ;
    k = PRODUCT ( ++i ) ;

    printf ( "\n%d %d", j, k ) ;
}
```

(d)
```
# define SEMI ;
main()
{
  int p = 3 SEMI ;
  printf ( "%d", p ) SEMI
}
```

[C] Attempt the following:

(a) Write down macro definitions for the following:

1. To test whether a character entered is a small case letter or not.
2. To test whether a character entered is a upper case letter or not.
3. To test whether a character is an alphabet or not. Make use of the macros you defined in (1) and (2) above.
4. To obtain the bigger of two numbers.

(b) Write macro definitions with arguments for calculation of area and perimeter of a triangle, a square and a circle. Store these macro definitions in a file called "areaperi.h". Include this file in your program, and call the macro definitions for calculating area and perimeter for different squares, triangles and circles.

(c) Write down macro definitions for the following:

1. To find arithmetic mean of two numbers.
2. To find absolute value of a number.
3. To convert a uppercase alphabet to lowercase.
4. To obtain the bigger of two numbers.

(d) Write macro definitions with arguments for calculation of Simple Interest and Amount. Store these macro definitions in a file called "interest.h". Include this file in your program, and use the macro definitions for calculating simple interest and amount.