Subject: COMPUTER PROGRAMMING IN "C" LANGUAGE **Credits: 4**

SYLLABUS

Introduction to 'C' Language, Structures of 'C' Programming. Constants and Variables, C Tokens, Operators, Types of operators, Precedence and Associativity, Expression , Statement and types of statements

Built-in Operators and function, Console based I/O and related built-in I/O function, Concept of header files, Preprocessor directives, Decision Control structures, The if Statement, Use of Logical Operators, The else if Clause

Loop Control structures, Nesting of loops , Case Control Structure, using Switch Introduction to problem solving, Problem solving techniques (Trial & Error, Brain storming, Divide & Conquer), Steps in problem solving (Define Problem, Analyze Problem, Explore Solution), Algorithms and Flowcharts (Definitions, Symbols), Characteristics of an algorithm

Simple Arithmetic Problems, Addition / Multiplication of integers, Functions, Basic types of function, Declaration and definition, Function call, Types of function, Introduction to Pointers, Pointer Notation, Recursion.

Storage Class, Automatic Storage Class , Register Storage Class, Static Storage Class, External Storage Class

Suggested Readings:

1. Mastering C by Venugopal, Prasad – TMH

2. Complete reference with C - Tata McGraw Hill

3. C – programming E.Balagurusamy - Tata McGray Hill

4. How to solve it by Computer - Dromey, PHI

Introduction to 'C' Language

1.1 History

1.2 Structures of 'C' Programming

1.3 Function as building blocks

1.4 Character set

1.5 Constants and Variables

1.6 C Tokens

1.7 Keywords

1.8 Identifiers

1.9 Data Types

1.10 Comments

Introduction to 'C' Language

'C' Is a general-purpose structured programming language. It is one of the most popular computer languages today. 'C' has also been called "system programming language", because it is useful for writing operating systems simultaneously it is equally powerful in writing major numerical, text and data-base processing applications.

'C' is referred to as middle level language. Before 'C' there were two types of languages used to program computers. One was the Assembly Language. Assembly language is a low level language because a programmer is working with the actual instructions which the computer will execute Assembly Language provides no control structure

The other was High Level Language. A high level language buffers a programmer from the computer. Its supplies various control structures, input and output commands which make programming easier and faster. There was a need for a language which could be more easily use, while at the same time provides nearly the same efficiency as assembly language.

C successfully combines the structure of a high level language and the power and efficiency of assembly language. Initially 'C' was used for creating system software. Today 'C' is used by programmers for virtually any programming task.

'C' was invented and first implemented by Dennis Ritchie on a DEC PDP-11 machine using UNIX Operating system. 'C' is the result of development process that started with an older language called BCPL (Basic Combined Programming Language), developed by Martin Richards. BCPL influence a language called B which was invented by Kan Thompson and which led to the development of 'C' in the 1970's.

'C' was written originally for programming under UNIX, which itself was later rewritten almost entirely in 'C'. Today, 'C' is running on a number of operating systems including MS-DOS. There is no single complier of 'C'. Many different organizations have written and implemented 'C' compliers, which differ from one other to a greater or lesser extend. Even for the same computer, there may be several different compilers, each with its own individual requirements.

Features of 'C'

- **General-purpose programming language**: 'C' was originally developed for system programming, but it is also used for writing commercial as well as scientific application.
- **Structured Programming Language**: 'C' provides the fundamental flow-control constructs require for well-structured programs. It allows developing well-defined, portable and easy-to-maintain programs.

- **Modularity**: We can break down the program code into functional units, develop independent codes and then integrate them to make complete application. The different modules can be present different files and can be compiled separately, there by facilitating team participation in development.
- **Portability**: 'C' programs written for one computer can executed on another with little or no modifications. C programs written to work under UNIX Operating system can be easily ported to work under MS-DOS operating system with little source code modifications. 'C' portability facilitates the use of new computer with the different operating system. 'C' programs can be easily ported to new hardware or operating system or both.
- **Code-reusability**: Besides the standard 'C' function library, a programmer can easily create his own function and use them over and over again in different application.
- **Ability to extend itself**: A 'C' program is basically a collection function(s) that are supported the 'C' library. We can continuously add our own function(s) to the 'C' library. With the availability of a large number of functions, the programming task becomes easier.
- **Limited number of key words**: There are only 32 key words in 'C'. The strength of 'C' lies in it's in –built function. Several standard functions are available with 'C' that is used for developing programs.

1.1 History

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:

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. 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.

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.

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.

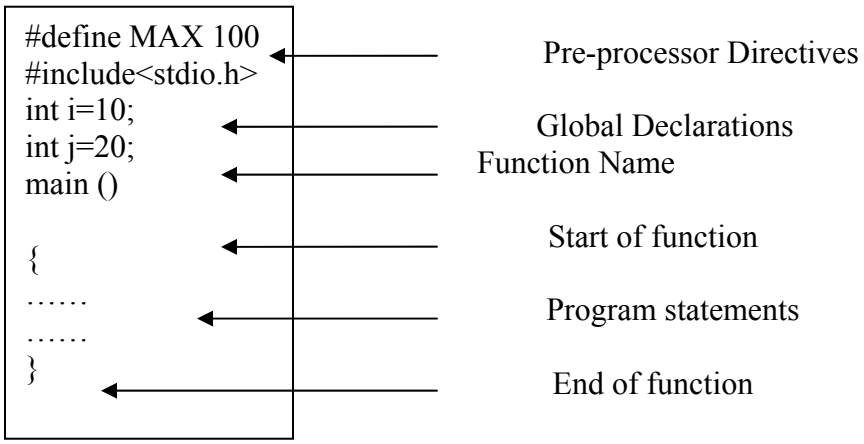
Mobile devices like cellular phones and palmtops are becoming increasingly popular. Also, common consumer devices like microwave oven, washing machines and digital cameras are getting smarter by the day. This smartness comes from a microprocessor, an operating system and a program embedded in this devices. 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.

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.

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.

1.2 Structures of 'C' Programming

'C' Program Structure



- **Pre-processor Directive(s)**

All 'C' compilers use as their first phase of compilation a pre-processor, which performs various manipulations on the source file before it is actually compiled.

Pre-processor directives are not actually a part of the 'C' language, but rather instructions from us to the compiler.
The #include directive tells the pre-processor to read in another file and include it with our program. The most commonly required header file is called stdio.h. All header files end with .h extention.

- **Global Declaration(s)**

There are two places where variables are declared. 1) inside a function or 2) outside all functions.

Variables declared outside all functions are called global variables and they may be accessed by any other function in the program. Global variables exist the entire time the program is executing.

- **Function(s)**

A function is a set of instructions aimed at achieving a pre-defined goal or objective.
A function in 'C' is similar to the user-defined-function of FoxPro.

Every 'C' program must have one main ( ) function. Based on requirements we may provide additions function(s) in a 'C' program.

The Function's body comprises of two parts.

I. Variable declarations
II. Executable statements.

All variables to be used in the function must be combined and put together before the first of exactable line. There is at least one executable statement in a 'C' function.

The two parts of a function 1) Variable declaration and 2) Executable statements must appear between the opening '{'and closing braces'}'.
The function execution begins at the opening brace and ends at the closing brace. The closing brace of the main ( ) function is the logical end of any 'C' program.
All statements in the declaration and executable parts end with a semicolon ';' known as statement terminator.

- **Programming style**

'C' is a case sensitive programming language, i.e., it will treat STG, stg and Stg differently.

'C' is a free-from programming language. The 'C' compiler does not care where on the line we begin typing. The developer can use this flexibility to make easy readable programs.

A statements in 'C' can start anywhere on a line and end anywhere on the line. The same statement can span several lines and the same line can contain several statements.
One or more white spaces will have to separate one word from another and where one white spaces is legal; any numbers of spaces are legal.
By white spaces we imply either a space or a tab or a carriage return.

The declaration

```
int x=50;
```

Can also be written as:

```
int
x
=
50
;
```

Similarly, we can group statements together on one line. The statements;

```
x=y;
n=y+1;
c=a+b;
```

Can be written in one line as:

```
x=y; n=y+1; z=a+b;
```

1.3 Function as building blocks

A function is a part of program statements, aimed at achieving a desired result.
A function could be written to calculate the larger of two numbers, print a desired message on the screen or calculate the interest payable, etc.

In 'C', as stated earlier, there are 32 key words only; rest of the working of 'C' is handled using function.

'C' provides its programmers two types of functions.

1. In-built functions
2. User-Defined function(UDF's)

We have already used two in-built function of 'C', the printf() function and the scanf() function for formatted input and formatted output operations. In the due course of this book, we wick study the other relevant in-built functions available to a programmer in 'C'.

The main() function used by us in all the programs written earlier is a compulsory function for every 'C' program to execute independently.

A 'C' source code file which does not contain the main() function, cannot be executed independently.

In the chapter we will be studying User Define Functions in 'C'.
The major difference between the in-built functions and user-Defined Functions is that, a programmer has to write a user-
Defined Function, whereas in-built functions are available to a programmer in the standard 'C' libraries. Due to this, the in-built functions are also known as library functions.

- **Advantages of Functions**

The strength of 'C' language lies in the fact that functions are easy to define and use. Functions facilitate the modular programming technique. Functions can be used to divide a large bulky program into functionally-independently modules or sub-programs that can be tested, debugged and maintained independently.

As discussed earlier, UDF's can be made a part of a library which, in turn, facilitates the usage of that function across various other 'C' programs. This feature, thereby, saves the time that one would have to take to record it for another application. This feature is known as "code-reusability" in general programming jargon.

Function Definition
The function in 'C' sre defined in the following format.
Definition of a 'C' function
Function-name([argument list})
[argument declaration];

```
{
[local variable declarations];
statement 1;
statement 2;
……
[return([expression])];
}
```

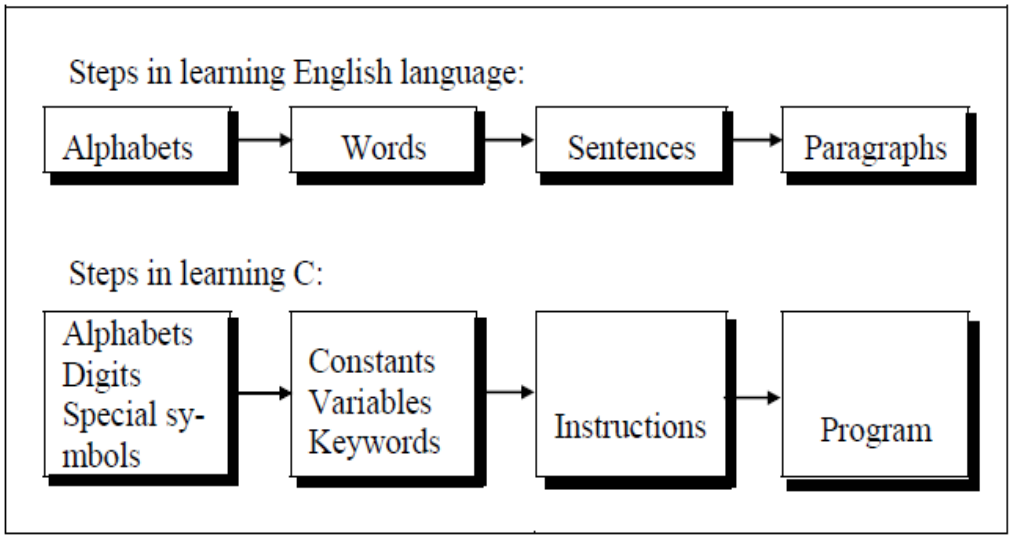
A function declaration in 'C' consist of the function name followed by a pair of open and close parentheses '()'. The function naming convention is the same as that of naming variables.

The parentheses can contain an optional list of arguments that are passed to the function. When arguments (parameters) are passed to a function they are defined before the body of the function (block of statement enclosed in braces).
The body of the function (executable block of statement) is enclosed with in braces. A function should have at least one executable statement. If no executable statement is present in a function, it is known as a dummy function. There is no limitation to the number of executable statements that can be included in the body of a function. The body of function can also have an optional declaration for the variables to be used by the function only. These variables are known as local variables.

A function can also have an optional return statement. The return statement can contain an optional expression or value that is to be returned back to the calling function.

1.4 Character set

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.



A character denotes any alphabet, digit or special symbol used to represent information.

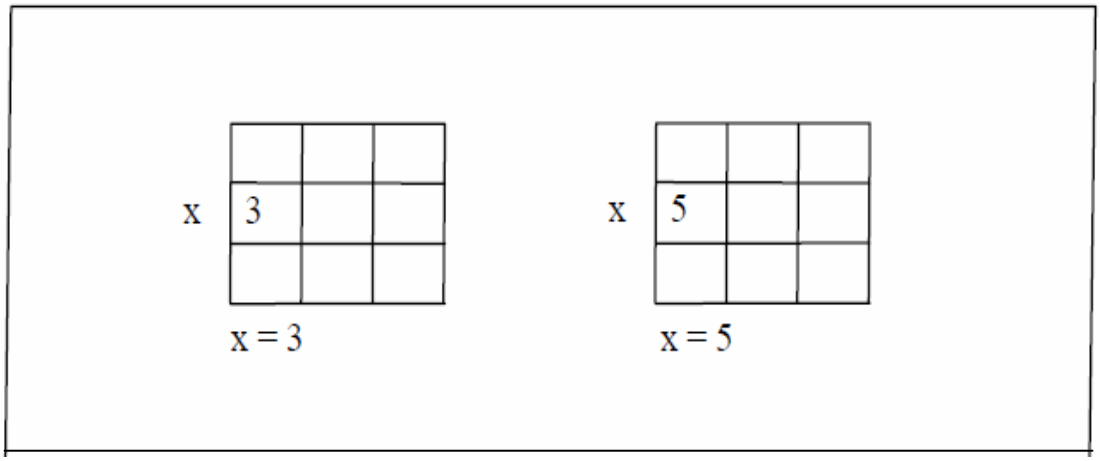
Alphabets	A, B, ….., Y, Z a, b, ……, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

1.5 Constants & Variables

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.

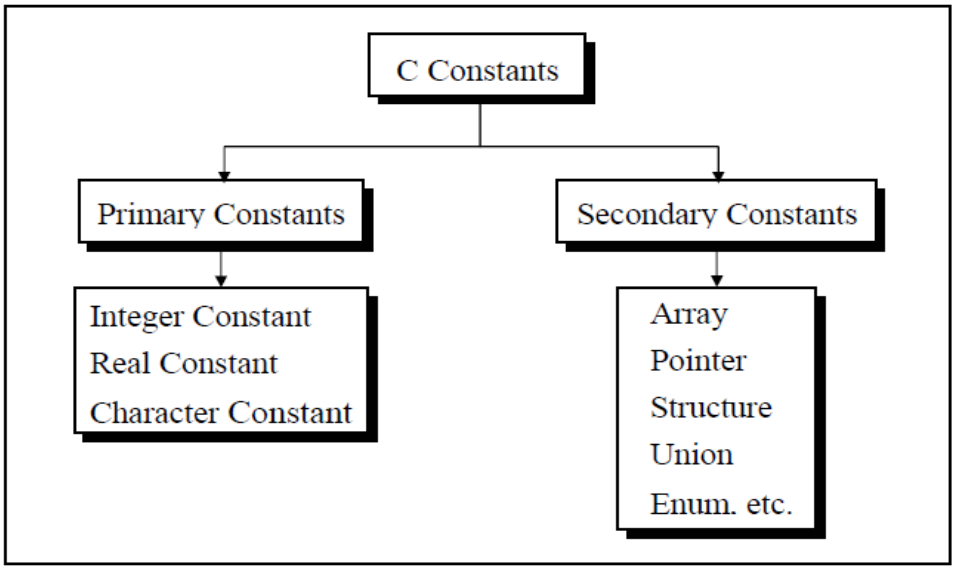
Here 3 is stored in a memory location and a name **x** is given to it. Then we are assigning a new value 5 to the same memory location **x**. This would overwrite the earlier value 3, since a memory location can hold only one value at a time.



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



Constants refer to values that do not change during the execution of a program.
'C' supports several types of constants.

Integer Constant

An integer constant refers to a sequence of digits.

There are three types of integer constants in 'C'.

1. Decimal
2. Octal
3. Hexadecimal

- **Decimal Integer Constant**

Decimal integer constants consist of set of digits, 0 through 9, preceded by an optional – or + sign. Embedded space, commas and non-digit characters are not permitted between digits of the integer constant.

Example

123
-987
0
+123456

The following are invalid integer constants:

15,000
$120
20,000
19.90

- **Octal integer constant**

An octal integer constant consists of any combination of digits between 0 and 7 with a leading 0.

Example

037
0
0657
0461

- **Hexadecimal integer constant**

A hexadecimal integer constant of any combination of digits between 0 and 9 and characters A to F or a to f. A hexadecimal integer constant is preceded by either 0x or 0X.

Example

0x2
0x2
0x9A
0X2f

Octal and Hexadecimal integers are very rarely used in programming.

The largest integer value that can be stored is dependent on the machine. For a 16bit machine, we can score 32767 and for a 32bit machine we can store 2,147,482,647 in an integer.
It is also possible to store larger constants on these machines by appending qualifiers U, L and UL to these constants.

Example

56876U	56876u	(unsigned integer)
1234569876UL	1234569876ul	(unsigned long integer)
9876543L	98765431	(long integer)

Real Constants

Integer numbers are inadequate to represent quantities that contain fraction parts. Number containing fraction part are called as real (or floating point) numbers.
These numbers are shown in decimal notation, having a whole number followed by a decimal point and the fraction part.A real number may also be expressed in exponential or scientific notation.

The general format for writing exponential or scientific is:

Mantissa e exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer with an optional plus or minus sign. The letter e separating the mantissa and the exponent can be written in either lower or uppercase. Since the exponent causes the decimal point to float, this notation is said to represent a floating point number. Exponential notation is used for representing numbers that are either very large or very small in magnitude.

Example

0.009879
1.23323
-0.9987
+2467.00
655
.95
-.77
+.88
0.89e4
12e-2
1.5E+9
-12.2E-5

Character Constants
Character constants are classified as:

1. Single Character Constants
2. Multiple Character or String Constants

- **Single Character Constants**

A single character constant (or simply character constant), contains a single character enclosed with a pair of single quotes. Character constants have an integer value associated with them known as ASCII
Values.

Example

'9'
'a'
's'
' '

- **Multiple Character or String Constants**

A string constant is a sequence of characters enclosed in a pair of double quotes. The characters may be letters, numbers, special characters or blank spaces.

Example

"Hello"
"STG International Limited"
"Good Luck"
"9+0"

A string constant does not have an associated ASCII value.

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. To be able to use variables to store data, we need to define them. Available is defined by associating a name and a data type with it.

- Variable Naming Conventions
- The name can be up to 31 characters long
- Can be made up of a-z, A-Z, 0-9 and the underscore '-'
- Cannot duplication a reserved or keyword in 'C'
- The first character cannot be a numeral
- All Variable names are case sensitive

After assigning suitable names to variables, we need to declare them to the compiler.

Declaration does the following two things:

i. It tells the compiler what the variable name is.
ii. It specifies what type of data the variable will hold.

The declaration of variable must be done before they are used in the program.

The syntax for declaring variable is as follows:

Data-type <variable1> [, variable2,……..variablen];

Variable1, variable2……variablen are names of variables.

If we want to declare multiple variables in the same line, we need to delimit them with commas. Every declaration statement must end with a semi-colon.

Example

```
int a;
float b;
double c;
char d;
```

int firstno, second no, thirdno, fourthno;
char firstinitial, secondinitial, lastinitial;

1.6 C Tokens

The tokens of a language are the basic building blocks which can be put together to construct programs. A token can be a reserved word (such as **int** or **while**), an identifier (such as **b** or **sum**), a constant (such as **25** or **"Alice in Wonderland"**), a delimiter (such as **{** or **;**) or an operator (such as + or =).

For example, consider the following portion of the program

```
main() {int a, b, sum;a = 14;b = 25;sum = a + b;printf("%d + %d = %d\n", a, b, sum);
}
```

Starting from the beginning, we can list the tokens (in **bold**) in order:

main - identifier

(- left bracket, delimiter

) - right bracket, delimiter

{ - left brace, delimiter

int - reserved word

a - identifier

, - comma, delimiter

b - identifier

, - comma, delimiter

sum - identifier

; - semicolon, delimiter

a - identifier

= - equals sign, operator

14 - constant

; - semicolon, delimiter

and so on. Thus we can think of a program as a 'stream of tokens', which is precisely how the compiler views it. So that, as far as the compiler is concerned, the above could have been written:

```
main() { int a, b, sum;
a = 14; b = 25; sum = a + b;
printf("%d + %d = %d\n", a, b, sum); }
```

The order of the tokens is exactly the same; to the compiler, it *is* the same program. To the computer, *only the order of the tokens is important*. However, layout and spacing are important to make the program more readable to

1.7 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.

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

Sample Programs in 'C'

As mentioned earlier, to write programs in 'C' under UNIX operating system, we can make use of any of the editors available to us like vi or ed.

In this section, we will take a look at a few sample programs in 'C', to understand

1. The 'C' program structure.
2. Declaration of variables
3. Assigning of values to variables.

Sample program 1

```
/* This program will print a welcome message on the screen*/
main ( ) /* main function starts */
{
        printf ("welcome to your first program in c");
} /* end of main function end of the program */
```

- Printf( )
  This function is used in 'C' language to display an output on the screen. The details of the function will be in the next chapter.

Sample program 2

```
/* This program introduces variable declaration and assignment of value */
main ( )
{
        int month;      /* Declaring variable */
        month = 12;     /* Assigning values */
        printf ("There are %d months in a year", month);
}
```

The output of the above program will be

There are 12 months in a year

- %d

  This is a format specifier in 'C', and is used with integer. We will learn in detail about the format specifiers in the next chapter.

Sample program 3

```
/* A simple program with two integer declarations */
main ( )
{
        int minutes = 60;
        int hours = 24;
        printf ("There are %d minutes in an hour and %d hours in a day", minutes, hours);
}
```

The output of the above program will be

There are 60 minutes in an hour and 24 hours in a day

In this program we have simultaneously used two format specifier in the same command line; C will treat them in the following manner.

Printf ("There are %d minutes in an hour and %d hours in a day \n", minutes, hours);

There are 60 minutes in an hour and 24 hours in a day.

The first format specifier is associated with the first constant or variable, the second with the second and so on.

Sample program 4

```
/* program to get input from the user and print the same*/
main ( )
{
        int firstno;
        printf("enter the value for the number: ");
        scanf ("%d", &firstno);
        printf ("\n you entered %d", firstno);
}
```

- Scanf ( )

  This function is used to get input from the user. The details of this function will be taken in the next chapter.

- \n

  Is the new line character, which forces a carriage return when used with the printf ( ) function.\n causes the output to be printed on the next line.

1.8 Identifiers

Identifier is the fancy term used to mean 'name'. In C, identifiers are used to refer to a number of things: we've already seen them used to name variables and functions. They are also used to give names to some things we haven't seen yet, amongst which are *labels* and the 'tags' of *structures*, *unions*, and *enums*.

The rules for the construction of identifiers are simple: you may use the 52 upper and lower case alphabetic characters, the 10 digits and finally the underscore '_', which is considered to be an alphabetic character for this purpose. The only restriction is the usual one; identifiers **must** start with an alphabetic character.

Although there is no restriction on the length of identifiers in the Standard, this is a point that needs a bit of explanation. In Old C, as in Standard C, there has **never** been any restriction on the length of identifiers. The problem is that there was never any guarantee that more than a certain number of characters would be checked when names were compared for equality—in Old C this was eight characters, in Standard C this has changed to 31.

So, practically speaking, the new limit is 31 characters—although identifiers **may** be longer, they must differ in the first 31 characters if you want to be sure that your programs are portable. The Standard allows for implementations to support longer names if they wish to, so if you do use longer names, make sure that you don't rely on the checking stopping at 31.

One of the most controversial parts of the Standard is the length of *external identifiers*. External identifiers are the ones that have to be visible outside the current source code file. Typical examples of these would be library routines or functions which have to be called from several different source files.

The Standard chose to stay with the old restrictions on these external names: they are not guaranteed to be different unless they differ from each other in the first six characters. Worse than that, upper and lower case letters may be treated the same!

The reason for this is a pragmatic one: the way that most C compilation systems work is to use operating system specific tools to bind library functions into a C program. These tools are outside the control of the C compiler writer, so the Standard has to impose realistic limits that are likely to be possible to meet.

There is nothing to prevent any specific implementation from giving better limits than these, but for maximum portability the six monocase characters must be all that you expect. The Standard warns that it views both the use of only one case and any restriction on the length of external names to less than 31 characters as obsolescent features. A later standard may insist that the restrictions are lifted; let's hope that it is soon.

1.9 Data Types

Data types are of three varieties—char, int, and float. It may seem odd to 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.

DATA TYPE	KEYWORD EQUIVALENT
Character	Char
Unsigned character	Unsigned char
Signed character	Signed char or char
Signed integer	Signed int or int
Signed short integer	Signed short int or short int or short
Signed short integer	Signed long int or long int or long
Unsigned integer	Unsigned int or unsigned
Unsigned short integer	Unsigned short int or unsigned short
Unsigned long integer	Unsigned long int or unsigned long
Floating point	Float
Double-precision floating point	Double
Extended double-precision floating point	Long double

1.10 Comments

All programming languages let you include comments in your programs. Comments can be used to remind yourself (and others) of what processing is taking place or what a particular variable is being used for. They can be used to explain or clarify any aspect of a program which may be difficult to understand by just reading the programming statements.

This is very important since the easier it is to understand a program, the more confidence you will have that it is correct. It is worth adding anything which makes a program easier to understand.

Remember that a comment (or lack of it) has absolutely no effect on how the program runs. If you remove all the comments from a program, it will run exactly the same way as with the comments.

Each language has its own way of specifying how a comment must be written. In C, we write a comment by enclosing it within /* and */, for example:

/* This program prints a greeting */

A comment extends from /* to the next */ and may span one or more lines. The following is a valid comment:

/* This program reads characters one at a time

and counts the number of letters found */

C also lets you use // to write one-line comments. The comment extends from // to the end of the line, for example:

a = s * s; //calculate area; store in a

In this series, we will use mainly one-line comments.

Operators

2.1 Types of operators

2.2 Precedence and Associativity

2.3 Expression

2.4 Statement and types of statements

0

Operators

2.1 Types of operators

Operators in 'c'
'c' provides a potent set of operators to perform different operation.
The operators available under 'c'can be broadly classified into the following categories

Assignment Operator
The assignment operator '='is used to assign values to variables in 'c'
Besides provides the assignment operator '=' using which we can assign values 'c' also provides shorthand assignment substitutes.that make the code compect.

- **Shorthand Assignment Notation**

The following tables illustrates shorthand Assignment Notations

Statement with simple assignment operator	Statement with shorthand operator
X=X+1	X+=1
X=X-1	x-=x
X=X*(n+1)	X*=+1
X=X/(n+1)	X/=n+1
X=X%Y	X%=Y

Arithmetic Operators

'C' provides all the basic arithmetic operators. These are listed below.

- **Addition Operator**

The addition operator '+' is used to add two or more numbers.

- **Subtraction Operator**

The subtraction operator '-' is used to find the difference between two or more numbers.

- **Multiplication operator**

The multiplication operator '*' is used to find the product of two or more numbers.

- **Division operator**

The division operator '/' is used to find the quotient of two numbers.

- **Module division operator**

The module division operator '%' is used to find the remainder of division of two numbers.

- **Increment and decrement operators**

Besides providing the standard arithmetic operator, 'C' also provides two other special arithmetic operators known as increment and decrement operators.

These operators increase or decrease the value of a variable without the use of the assignment operator.

The increment and decrement operators are represented as:

++	Increment operator
--	Decrement operator

Both the increment and decrement operators can be present before the variable or after the variable.

The result stored in the variable would be different when the increment or decrement operator is present before the variable or when the increment or decrement operator is present after the variable.

Example

Consider a variable 'a' containing an initial value '5', the statement written in 'C' to achieve this would be:

```
int a=5;      /* declaring a variable a of integer type and initializing the value to 5 */
```

We can use the increment operator '++' in the following manner:

```
a++;
```

or

```
++a;
```

The above statement a++; and ++a; are equal to a = a+1; or (a +=1);

In this case, both the statements would result in the value of 'a' being incremented by '1' 'a' would now have '6' stored in it after the increment operator is used.

In a similar manner we can use the decrement operator '-' to decrement the value of 'a' by one.

When we use the increment or decrement operator independently with only the variable in the statement (like we have used in the above example), there is no difference in the result of using the increment (or decrement) operator before the variable (pre-fixed) or after the variable (post-fix). Same is true for decrement operator.

The increment and decrement operators do not need to follow the variable; they can precede it. Although the effect on the variable is the same, the position of the operator does affect when the operation is performed.

Operators

2.1 Types of operators

Operators in 'c'
'c' provides a potent set of operators to perform different operation.
The operators available under 'c'can be broadly classified into the following categories

Assignment Operator
The assignment operator '='is used to assign values to variables in 'c'
Besides provides the assignment operator '=' using which we can assign values 'c' also provides shorthand assignment substitutes.that make the code compect.

- **Shorthand Assignment Notation**

The following tables illustrates shorthand Assignment Notations

Statement with simple assignment operator	Statement with shorthand operator
X=X+1	X+=1
X=X-1	x-=x
X=X*(n+1)	X*=+1
X=X/(n+1)	X/=n+1
X=X%Y	X%=Y

Arithmetic Operators

'C' provides all the basic arithmetic operators. These are listed below.

- **Addition Operator**

The addition operator '+' is used to add two or more numbers.

- **Subtraction Operator**

The subtraction operator '-' is used to find the difference between two or more numbers.

- **Multiplication operator**

The multiplication operator '*' is used to find the product of two or more numbers.

- **Division operator**

The division operator '/' is used to find the quotient of two numbers.

- **Classroom Exercise**

  What are the advantages of mixed mode?

Relational operators are used to compare two quantities. Comparison is made to take a certain decision depending upon the relationship between the two quantities.

An expression containing a relational operator is known as a relational expression. It always evaluates to either true (non zero) or false (0).

'C' supports six relational operators.

- **Equal To**

To compare two quantities for equality, the "equal to" relational operator is used.

Notice here that we have used a double equal to sign '==' to represent the "equal to" relational Operator. In 'C', if we wish to compare two quantities for equality, a '==' sign is used. We cannot use a single equal to '=' sign, because it is the assignment operator and would assign values rather than comparing it.

- **Less Than**

The "less than" relational operator is used to compare if the quantity on the left-hand side of the expression is similar than the quantity on the right-hand side of the expression.
'<' Symbol is used represent the "less than" relational operator.

- **Less Than or Equal To**

The "less than or equal to" relational expression is used to determine if the quantity on the left-hand side of the expression is smaller than or equal to the quantity on the right-hand side of the expression.

'<=' Symbol is used to represent the "greater than " relational operator.

- **Greater than**

The "greater than " relational operator compares to find if the quantity on the left-hand side of the expression is bigger than the quantity on the right-hand side.
'>' Symbol is used to represent the "greater than" relational operator.

- **Greater Than or Equal To**

The "greater than or equal to " relational operator compares if the quantity on the left-hand side of the expression is bigger or equal to the quantity on the right-hand side of the expression.

'>=' Symbol is used to represent " greater than or equal to " relational operator.

- **Not Equal To**

The "not equal to" relational operator is used to compare if the quantity on the left-hand side of the expression is not the same as the quantity on the right-hand side of the expression.

'!=' Symbol is used to represent "not equal to" relational operator.

- **Classroom Exercise**

  Can we use the assignment operation in place of the equal to operator? If not why?

Logical Operators

Logical operators are used to combine two or more relational expression into one relational expression. An expression of this kind, which combines two or more relational expressions, is termed as logical expression or compound relational expression.

Similar to a simple relational expression, a logical expression also evaluates to either true (non zero) or false (0).

- **AND**

The AND operator when used with a relational expression, ensures that the logical expression evaluates to true if and only if all the conditions are true, otherwise the logical expression evaluates to false.

The AND operator is represented by a double ampersand symbol '&&'.

- **OR**

When we use the OR operator expression, the expression evaluates to true if at least one of the relational expressions is true, otherwise, the logical expression evaluates to false.
The OR operator is represented by a double pipe symbol "||".

- **NOT**

NOT operator is used to reverse (or negate) truth value of an expression or expressions.

The NOT operator is represented by an exclamination mark symbol '|'.

Following table shows the truth table for the above logical operators.

P	q	p&&q	P||q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Unary Operator

These are operators that can function without the assignment or relational operator is used. With the case of the Unary operators, we use neither.

A unary operator works on one quantity.

- The minus '-' symbol is used to represent Unary Minus operator.

Example

```
-3
-4.2
-89UL
```

- **Logical NOT**

This unary operator is used for inverting the value of the operand. It only works on integer type of data. It is represented by the exclamation mark '!'.

Example

```
! (a==b)
! x
```

- **Address**

This unary operator is used to find the address of a memory variable. The details of the operator will be taken up in the chapter on pointers.

It is represented using the ampersand symbol '&'.
Example

```
&a
```

- **Indirection**

This operator returns contents of a location whose address is the operand. The details of this operator will also be taken up in the chapter on pointers.

This operator only works on the address of the memory location for a variable.

It is represented by the asterisk symbol '*'.

Example
```
*b
*(a+1)
```

- sizeof( )

This operator is used to return the size of the operand in bytes.

Example
```
sizeof (a)
sizeof (int)
```

The number of bytes taken up by a variable is directly dependent on the machine that is being used (16 bit machine or 32 bit machine). An integer takes up 2 bytes on a 16 bit machine and 4 bytes on a 32 bit machine. Therefore, the value returned by this operator will also depend upon the type of machine in use.

- **Classroom Exercise**
  How does 'C' evaluates! (a==b) statement?

Binary Operator
Binary operators are the operators that require at least two quantities to function.

These operators are normally associated with the assignment, relational and logical operations.

- Types of Binary Operators

Addition	+	Subtraction	-
Multiplication	*	Division	/
Modulo Division	%	Assignment	=
Equal To	==	Less than	<
Less than equal to	<=	Greater than	>
Greater than equal	>=	Not equal to	!=
Logical AND	&&	Logical OR	||
Bitwise AND	&	Bitwise-exclusive OR	^
Bitwise OR	|	Left sift	<<
Right shift	>>	One's complement	~

- **Classroom Exercise**

  Can module division operator be used in place of division operator? If yes how?

Ternary Operator

Ternary Operator is also known as the conditional operator.
This operator takes the form:

<test-operand> ? operand-2 : operand-3

The test-operand is an integer / expression. The other two operands can be any values or variables, so long as they are of the same type.

If the test-operand is true, operand-2 is returned and operand-3 not evaluated. Otherwise, operand-3 is returned and operan-2 is not evaluated.

The ternary operator is represented by the question mark '?'.

Example

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

Comma Operator

The comma operator can be used to link the related expressions together.

A comma-linked list of expression is evaluated left-to-right.

The value of right-most expression is the value of the combined expression.

Example

```
Number  = (a=50, b=100, a+b);
```

In the above statement, 'a' is assigned the value 50, then 'b' is assigned the value 100 and finally, 'number' is assigned the value 150 as a result of the addition of 'a' and 'b'.

The comma operator is extensively used in the for loop, which we will discuss in a later chapter.
The comma operator falls under the category of binary operators.

Types Cast Operator

'C' permits mixing of constants and variable s of different types in an expression, but during evaluates it follows a definite rule of type conversion.

If the operands are of different types, the lower types types is automatically converted to the higher types before the operation proceeds. This is known as Automatic Type Conversion.

In an expression, if different types of operands are used on the right-hand side of the expression, the 'C' compiler automatically converts the lower data types to higher.

For example, in an expression we have got integer and float data types on the right-hand side of the expression, 'C' will automatically convert both the quantities to float and then evaluate the expression.

However, the final result of such an expression is converted to the type of the variable on the left-hand side already defined in the program.

For example, the variable on the left-hand side is an integer, and we have integer and float on the right-hand side, then during evaluation, 'C' first converts the integer on the right-hand side to float, evaluates the expression and the final result is again converted into an integer and stored in the variable on the left-hand side.

We should keep the following things in mind for automatic type conversion.

1. float to int causes truncation of the fractional part.
2. double to float causes rounding of digits
3. long int to int caused dropping of the excess higher order bits

'C' provides us with automatic type conversion for different data types on the right-hand side of the expression.

However, there are situations when we want to force a type conversion in a way that is different from the automatic conversion.

Consider, for example, the calculation of the ratio of boys in a school, we would like the expression as:

Ratio = no_of_girls / no_of_boys

Assume, the 'no_of *girls' and the 'no*_of_boys' was declared as integer and 'ratio' as float. In the program, the decimal part of the result of the division would be lost and the ratio would represent a wrong figure.

How then do we solve this problem….?

We can solve this problem by converting locally one of the variables to float, as

Ratio = (float) no_of_girls / no_of_boys

The operator (float) converts 'no_of_girls' to floating point for the purpose of evaluation of the expression. Then using automatic conversion, the division is performed in floating point mode, thus retaining the fraction part, which is stored in 'ratio' defined as a 'float' variable.

It is important to note that the operator (float) converts the variable no_of_girls to floating point for the purpose of evaluation only. The value of the variable does not change and also the type of no_of_girls remains as int in the other parts of the program.

The process of such a local type conversion is known as casting a value.

The general format for the cast operation is :

(type_name) expression

type_name is one of the standard data types available in 'C'.

The expression may be a constant, variable or expression.

The cast operator belongs to the category of unary operators.

2.2 Precedence and Associativity

Each operator in 'C' has a precedence associated with it. This is used to determine how an expression involving more than one operator is evaluated

'C' provides distinct levels of precedence. An operator at a higher level of precedence than other will get evaluated first.

In case, the two operators are at the same level of PROCEDENCE, THEY ARE EVALUATED EITHER FROM LEFT TO RIGHT OR RIGHT TO LEFT. This is known as the associatively property of an operator.

SYMBOL	DESCRITION	ASSOCIATIVITY
++	Post-increment	Left to right
--	Post-Decrement	Left to right
()	Function Call	Left to right
[]	Array element reference	Left to right
->	Pointer to structure member	Left to right
.	Structure or union member	Left to right
++	Pre-increment	Right to left
--	Pre-decrement	Right to left
!	Logical NOT	Right to left
~	Bitwise NOT (one's Complement)	Right to left
+	Unary Plus	Right to left
-	Unary Minus	Right to left
*	Indirection (pointer reference)	Right to left
7	Address	Right to left

Sizeof()	Size of an object	Right to left
(type)	Cast conversion	Right to left
->*	Pointer to member	Left to right
*	Multiplication	Left to right
/	Division	Left to right
%	Modulo Division	Left to right
+	Addition	Left to right
-	Subtraction	Left to right
<<	Left Shift	Left to right
>>	Right Shift	Left to right
<	Less than	Left to right
<=	Less than equal to	Left to right
>	Greater than	Left to right
>=	Greater than equal to	Left to right
==	Equality	Left to right
!=	Inequality (NOT equal to)	Left to right
&	Bitwise AND	Left to right
^	Bitwise Exclusive OR (XOR)	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?	Conditional expression	Right to left
=,*=,/=,+=,-=,&=,^=,|=,<<=,>>=	Assignment operators	Right to left
,	Comma operator	Left to right

2.3 Expression

Arithmetic Expressions

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression	**C Expression**
a x b – c	a * b – c
(m + n) (x +	(m + n) * (x +

y)	y)
(ab / c)	a * b / c
$3x^2$ +2x + 1	3*x*x+2*x+1
(x / y) + c	x / y + c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Example of evaluation statements are

x = a * b – c

y = b / c * a

z = a – b / c + d;

The following program illustrates the effect of presence of parenthesis in expressions.

```
.main ()
{
float a, b, c x, y, z;
a = 9;
b = 12;
c = 3;
x = a – b / 3 + c * 2 – 1;
y = a – b / (3 + c) * (2 – 1);
z = a – ( b / (3 + c) * 2) – 1;
printf ("x = %fn",x);
printf ("y = %fn",y);
printf ("z = %fn",z);
}
```

output

x = 10.00
y = 7.00
z = 4.00

Precedence in Arithmetic Operators

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

High priority * / %

Low priority + -

Rules for evaluation of expression

- First parenthesized sub expression left to right are evaluated.
- If parenthesis is nested, the evaluation begins with the innermost sub expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
- The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When Parenthesis are used, the expressions within parenthesis assume highest priority.

Type conversions in expressions

Implicit type conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance. This automatic type conversion is know as implicit type conversion. During evaluation it adheres to very strict rules and type conversion. If the operands are of different types the lower type is automatically converted to the higher type before the operation proceeds. The result is of higher type.

The following rules apply during evaluating expressions

All short and char are automatically converted to int then

1. If one operand is long double, the other will be converted to long double and result .will be long double.
2. If one operand is double, the other will be converted to double and result will be double.

3. If one operand is float, the other will be converted to float and result will be float.
4. .4. If one of the operand is unsigned long int, the other will be converted into unsigned long int and result will be unsigned long int.
5. If one operand is long int and other is unsigned int then
   - **a.** If unsigned int can be converted to long int, then unsigned int operand will be converted as such and the result will be long int.
   - **b.** Else Both operands will be converted to unsigned long int and the result will be unsigned long int.
6. If one of the operand is long int, the other will be converted to long int and the result will be long int. .
7. If one operand is unsigned int the other will be converted to unsigned int and the result will be unsigned int.

Explicit Conversion

Many times there may arise a situation where we want to force a type conversion in a way that is different from automatic conversion.

Consider for example the calculation of number of female and male students in a class

.....female_students
Ratio=.....----------
.....male_students

Since if female_students and male_students are declared as integers, the decimal part will be rounded off and its ratio will represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below.

Ratio = (float) female_students / male_students

The operator float converts the female_students to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed by floating point mode, thus retaining the fractional part of the result. The process of such a local conversion is known as explicit conversion or casting a value. The general form is

(type_name) expression

2.4 Statement and types of statements

A C Program is made up of statements. Statement is a part of your program that can be executed. In other words every statement in your program alone or in combination specifies an action to performed by your program. C provides variety of statements to help you attain any function with maximum flexibility and efficency. One of the reason for popularity of C is because of the extreme power provided to programmer in C due to it rich and diverse set of statements define in

C. For becoming a top notch programmer you must have clear understanding of the C statments and the situations where staments in C are applicable.

Types Of Statements

Statements in C are categorized into following types-

1. Selection/Conditional Statement - They decide the flow of statements on based on evaluation of results of conditions. if - else and switch statements come under this category.
2. Iteration Statements - These are used to run a particular block statements repeatedly or in other words form a loop. for, while and do-while statements come under this category.
3. Jump Statements - The are used to make the flow of your statements from one point to another. break, continue, goto and return come under this category.
4. Label Statements - These are used as targets/waypoints for jump and selection statements. case (discussed with switch) and label (discussed with goto) come under this category
5. Expression Statements - Any valid expression makes an expression statement. To study expressions in detail click here .
6. Block Statement - A group of statements which are binded together to form a logic are called block statements. Block statement begins with a { and ends with a }.

Built-in Operators and function

built-in I/O function

3.1.1 printf( )

3.1.2 scanf( )

3.1.3 getch( )

3.1.4 getchar( )

3.1.5 putchar( )

3.2 Concept of header files

3.3 Preprocessor directives :

3.3.1 #include

3.3.2 #define

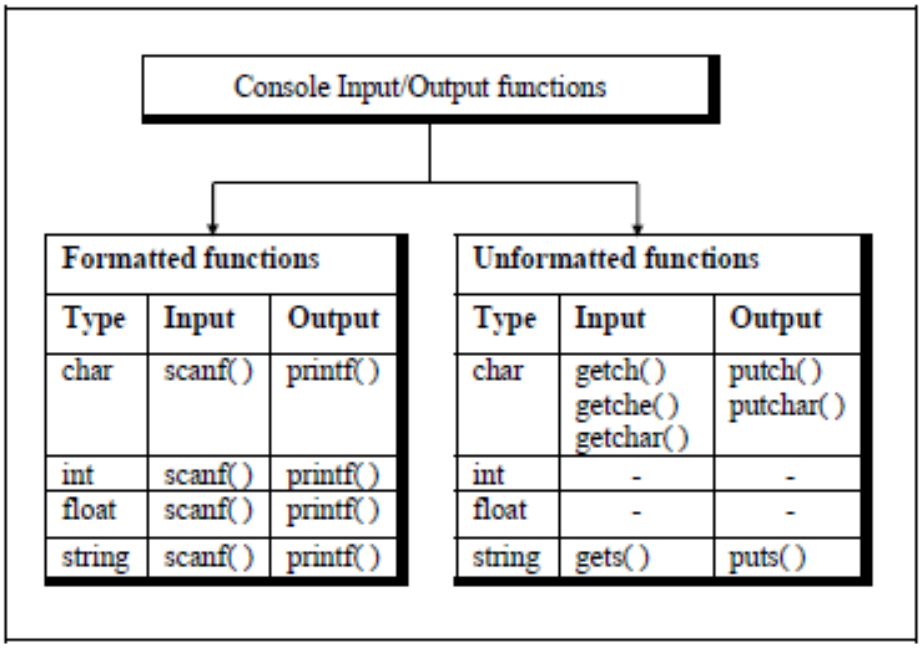
Built-in Operators and function

3.1 Console based I/O and related built-in I/O function

There are numerous library functions available for I/O. These can be classified into three broad categories:

(a) Console I/O functions - Functions to receive input from keyboard and write output to VDU.
(b) File I/O functions - Functions to perform I/O operations on a floppy disk or hard disk.

The screen and keyboard together are called a console. Console I/O functions can be further classified into two categories— formatted and unformatted console I/O functions. The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points, etc. can be controlled using formatted functions. The functions available under each of these two categories are shown in Figure. Now let us discuss these console I/O functions in detail.



the functions **printf( ),** and **scanf( )** fall under the category of formatted console I/O functions. These functions allow us to supply the input in a fixed format and let us obtain the output in the specified form

3.1.1 printf( )

Formatted output in 'C' is done using the printf () function. We already used the printf () function now we will discuss all the options that are available with this function, and ways of using them.

The syntax of printf () function is:
Printf("format specifiers",<variable1> [<,variable2>…….<,variablen>]);

"format specifier" consists of

1. Characters that will be printed on the screen as they appear.
2. Format specification that define the output format for display of each item
3. Escape sequence characters

"Variable1…..variablen" are the variable whose values are formatted and printed according to the specifications given in the format specifier.
A simple format specification has the following form:
%n.d data-type
n is an integer that specified the total number of columns for the output value and d is another integer number that specifies the number of digits to the right of the decimal point or the number of characters to be printed from a string. Both n and d are optional.
The printf(0 function does not automatically add the new line character at the end of the display. Therefore, we can use multiple printf () statements to display a single like of output. A new line can be introduced using the printf() function with help of the new line character '\n'.

Types of formatted output

- Integer output
- Real number output
- Character or string output
- Mixed-mode output

Integer output

The format specifier used for printing an integer number is
%nd
n specifies the minimum field width for the output.

In case, the number being printed is greater that the specified width, the entire number will be printed 'C' overrides the minimum field width definition for this case.

The numbers are printed right-justified by default leading blanks will appear for a number of lesser width than the width specified.

Example
Consider the following declaration
Int a=1234;
We will now see how printf(0will display the output under different formats
Printf("%d",a);
The output will be:

1	2	3	4

printf("%d", a);
the output will be :

		1	2	3	4

printf("%2d",a);
the output will be:

1	2	3	4

printf("%-6d",a);
the output will be :

1	2	3	4		

printf("%06d", a);
the output will be

0	0	1	2	3	4

It is possible to force left-justified printing of numbers using the minus '-'sign directly after the % sign. It is possible to replace leading blanks with zeros.

To print long int. we specify used %id instead of %d in the format specification.

Real number output

The format sepecifier used to display real number in floating point notation with decimals is:
%n.df

The integer value 'n' is the maximum number of positions that are to be used for the display of the whole number part of the value and the integer'd' indicates the number of digits to be displayed after the decimal point (precision).

The value pointed i.e. rounded off to 'd' decimal points and right-justified in a field of 'n' coloumns.

Example
Consider the following assignment
Float a = 23.7654
We will now see how printf () will display the output under real number format
Printf("%7.4f",a);
The output will be:

2	3		7	6	5	4

printf("%7.2f",a);
The output will be :

		2	3		7	7

The floating point number will be displayed in the form
[-]xxx.yyy

Mixed mode output

Similar to the mixed-mode operations with the scanf() function, a programmer can use . Mixed mode operations with the printf() function.
Format specfiers available with printf() function are given in the table below

Code	Description
%c	Single character
%d	Decimal integer
%e	Floating point value in exponent form
%f	Floating point value without exponent
%g	Floating point value either e or f type depending on value
%i	Signed decimal integer
%o	Octal integer; without leading zeros
%s	String
%u	Unsigned decimal integer
%x	Hexadecimal integer; without leading 0x
%l	Long
%uh	Unsigned short
%ul	Unsigned long

Escape Sequence
Escape sequence is the name given to backslash constants that are available in 'C'. Each of these contents is interpreted as one character though they contain two characters.

These two character combination backslash '\' and another character are known as escape sequence.
The table below lists the various backslash constants as available in 'C'

Constants	Description
'\a'	Bell
'\b'	Back space
'\f'	Form space
'\n'	New line
'\r'	Carriage return
'\t'	Horizontal Tab
'\v'	Vertical tab
'\''	Single quote
'\''''	Double quote
'\?	Question mark
'\\'	Backslash
'\0'	Null

The escape sequence is used with output function.

3.1.2 scanf( )

Formatted input in 'C' is done using the scanf () function. We have already used the scanf () function now we will discuss in detail all the options that are available with this function and ways of using them

The syntax of scanf () function is

scanf ("format specifiers", <argl> [<, arg2>…….<, argn>]);

"Format specifier" specifies the data type for the variable to be read.

Arg1….argn specifies the address of the variable where the keyed in data will be stored.

"format specifier" and variable list is separated by commas.
"format specifier" consists of

1. Conversion character '%'
2. A data type character (or type specifier)
3. An optional number specifying the input width.

- Types of formatted input
  - Integer input
  - Real number input
  - Character input
  - Mixed-mode input

Integer input

The format specifier for integer input is written as:
%nd

a specifies to scanf()function that an integer has to be read.

n specifies the input width for the integer data to be read.

Example
```
int a, b;
Scanf ("%d %d", &a, &b);
```

Real number input

Unlike integer numbers, the field width of real numbers is not specified and therefore scanf() reads real numbers using the simple specification '%f' for both integer part as well as fraction part. %f is also used for reading decimal notations as well as exponential notations.

Example

```
Float a, b, c;
Scanf ("%f %f %f", &a, &b, &c);
```

On the data line, if following input is given.
123.45 56:78 E-1 9012

scanf () will assign 123.45 to "a". 5.678 to"b" and 9012 to "c".
If we wish to read number of double data, we use '%if' instead of simple '%f'/

Character Input

We have already learnt how to read a single character from the keyboard using the getchar () function. The same can be achieved using the scanf () function. Besides reading a single character from the keyboard, scanf () function can also be used to read a string made up of more than one character from the keyboard.

The format specifier used with scanf () function to read a character or a string is
%nc for characters
%ns for string

%nc reads data from the keyboard into variable defined as char.
%ns reads data from the keyboard into a character array defined earlier in the program.

In this section we will limit ourselves to the of characters only.

The detail of reading string would be taken up in a latter chapter on arrays and strings.

Example
```
Char a, b;
scanf ("%c %c", &a, &b);
```

Mixed-mode input

It is possible to use scanf () statement to input variable of different data types. Appropriate format specifiers should be used for different data types.

When an attempt is made to read an item that does not match the type expected, scanf () immediately terminates any further reading and returns the values already read.

Example
```
Int a;
Float b;
Char x;
scanf ("%d %f %c", &a, &b, &c);
```

3.1.3 getch( )

getch() is used to get a character from console but does not echo to the screen.

Library:
<CONIO.H>

Declaration:
int getch(void);

Example Declaration:
```
char ch;
ch = getch(); (or ) getch();
```

Remarks:
getch reads a single character directly from the keyboard, without echoing to the screen.

Return Value:
This function return the character read from the keyboard.

Example Program:
```
void main()
{
char ch;
ch = getch();
printf("Input Char Is :%c",ch);
}
```

Program Explanation:
Here, declare the variable **ch** as char data type, and then get a value through ***getch()*** library function and store it in the variable ***ch*** .And then, print the value of variable ***ch***. During the program execution, a single character is get or read through *the* ***getch()***. The given value is not displayed on the screen and the compiler does not wait for another character to be typed. And then, the given character is printed through the **printf** function.

3.1.4 getchar( )

To read a character from the keyboard makes use of the getchar() function.

The syntax of getchar () function is:
Variable_name = getchar ();

"variable_name" is a valid 'C' variable that has been declared of char type.

When the above statement is executed, the program execution waits for a key press and then assigns the key pressed to the getchar () function.
As we have used the getchar () function on the right-hand side of the expression, the value returned by the getchar () function i.e., the key that was pressed is assigned to the variable_name on the left-hand side of the expression.

Example

```
Char string;
String = getchar ();
```

The above program segment will assign 'A' to string, if the user pressed A from the keyboard.
There is another one worth to be mentioned

clrscr(); :- This is use to clear the output screen i.e console

suppose you run a program and then alter it and run it again you may find that the previous output is still stucked there itself, at this time clrscr(); would clean the previous screen.

One more thing to remember always use clrscr(); after the declaration like

int a,b,c;

float total;

clrscr();

3.1.5 putchar( )

Analogous to the getchar () function for character input 'C' provides putchar () function for character output.
The syntax of putchar () function is:

Putchar (variable_name)

"variable_name" is a valid 'C' variable that has been defined as char type.

When this statement is executed, it displays the character contained in the variable_name on the terminal.

Example

```
Char choice = 'N';
Putchar (choice);
```

The above program display 'N' on the screen.

Example

```
/* sample program to make use of getchar() and putchar() functions
*/
#include <stdio.h>
main ()
{
char choice;
printf ("please enter a character from the keyboard :");
choice = getchar ();
printf ("\n you typed the following character :");
putchar (choice);
putchar ('\n');
}
```

putchar ('\n'); will cause the cursor on the screen to move to the beginning or the next line.

The printf () and scanf () functions can also be used for character input-output operations.

3.2 Concept of header files

A header file is a file containing C declarations and macro definitions to be shared between several source files. You request the use of a header file in your program by *including* it, with the C preprocessing directive `#include'.

Header files serve two purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

In C, the usual convention is to give header files names that end with `.h`. It is most portable to use only letters, digits, dashes, and underscores in header file names, and at most one dot.

3.3 Preprocessor directives :

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash sign (#). The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements.

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

3.3.1 #include

Both user and system header files are included using the preprocessing directive `#include'. It has three variants:

#include <*file*>

This variant is used for system header files. It searches for a file named *file* in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option `-I' The option `-nostdinc' inhibits searching the standard system directories; in this case only the directories you specify are searched.

The parsing of this form of `#include' is slightly special because comments are not recognized within the `<...>'. Thus, in `#include <x/*y>' the `/*' does not start a comment and the directive specifies inclusion of a system header file named `x/*y'. Of course, a header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate.

The argument *file* may not contain a `>' character. It may, however, contain a `<' character.

`#include "file"`

This variant is used for header files of your own program. It searches for a file named *file* first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to. (If the `-I-' option is used, the special treatment of the current directory is inhibited.)

The argument *file* may not contain `"' characters. If backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, `#include "x\n\\y"' specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

`#include` *anything else*

This variant is called a *computed #include*. Any `#include' directive whose argument does not fit the above two forms is a computed include. The text *anything else* is checked for macro calls, which are expanded. When this is done, the result must fit one of the above two variants--in particular, the expanded text must in the end be surrounded by either quotes or angle braces.

This feature allows you to define a macro which controls the file name to be used at a later point in the program. One application of this is to allow a site-specific configuration file for your program to specify the names of the system include files to be used. This can help in porting the program to various operating systems in which the necessary system header files are found in different places.

How `#include' Works

The `#include' directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the `#include' directive. For example, given a header file `header.h' as follows,

```
char *test ();
```

and a main program called `program.c' that uses the header file, like this,

```
int x;
#include "header.h"

main ()
{
  printf (test ());
}
```

the output generated by the C preprocessor for `program.c' as input would be

```
int x;
char *test ();

main ()
{
  printf (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file.

However, a comment or a string or character constant may not start in the included file and finish in the including file. An unterminated comment, string constant or character constant in an included file is considered to end (with an error message) at the end of the file.

It is possible for a header file to begin or end a syntactic unit such as a function definition, but that would be very confusing, so don't do it.

The line following the `#include' directive is always treated as a separate line by the C preprocessor even if the included file lacks a final newline.

3.3.2 #define

This comes in two flavours:

#include <filename>
#include "filename"

Both of which cause a new file to be read at the point where they occur. It's as if the single line containing the directive is replaced by the contents of the specified file. If that file contains erroneous statements, you can reasonably expect that the errors will be reported with a correct file name and line number. It's the compiler writer's job to get that right. The Standard specifies that at least eight nested levels of # include must be supported.

The effect of using brackets <> or quotes " " around the filename is to change the places searched to find the specified file. The brackets cause a search of a number of implementation defined places, the quotes cause a search of somewhere associated with the original source file. Your implementation notes must tell you the specific details of what is meant by 'place'. If the form using quotes can't find the file, it tries again as if you had used brackets.

In general, brackets are used when you specify standard library header files, quotes are used for private header files—often specific to one program only.

Although the Standard doesn't define what constitutes a valid file name, it does specify that there must be an implementation-defined unique way of translating file names of the form xxx.x (where x represents a 'letter'), into source file names. Distinctions of upper and lower case may be ignored and the implementation may choose only to use six significant characters before the '.' character.

You can also write this:

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

to get the same effect as

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

but it's a rather roundabout way of doing it, and unfortunately it's subject to implementation defined rules about how the text between < and > is treated.

It's simpler if the replacement text for NAME comes out to be a string, for example

```
#define NAME "stdio.h"

#include NAME
```

There is no problem with implementation defined behaviour here, but the paths searched are different, as explained above.

For the first case, what happens is that the token sequence which replaces NAME is (by the rules already given)

```
<
stdio
.
h
>
```

and for the second case

```
"stdio.h"
```

The second case is easy, since it's just a *string-literal* which is a legal token for a # include directive. It is implementation defined how the first case is treated, and whether or not the sequence of tokens forms a legal *header-name*.

Finally, the last character of a file which is being included must be a plain newline. Failure to include a file successfully is treated as an error.

Decision Control structures

4.1 The if Statement
4.2 The Real Thing
4.3 Multiple Statements within if
4.4 The if-else Statement
4.5 Nested if-else
4.6 Forms of if
4.7 Use of Logical Operators
4.8 The else if Clause
4.9 Nested if-else
4.10 Use of Logical Operators
4.11 The else if clause

Decision Control structures

In C programming the instructions are executed sequentially, by default. At times, we need a set of instructions to be executed in one situation and an another set of instructions to be executed in another situation. In such cases we have to use decision control instructions. This can be achieved in C using;

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

4.1 The if Statement

Before we understand Decision control structures its worth revising a concept related to selection constructs, iterations and a 'C' program in general. The concept is that of statements and blocks.

A statement in 'C' is an expression or a function calling that if terminated by a semi-colon ';'.
Example

```
Main ( )
{
        breadth = 2;
        printf ("welcome to STG");
        area = length *breadth;
}
```

While using selection constructs and iterations in 'C', we group multiple statements into blocks using braces. There is never a semicolon after the right (closing) brace that ends a block.

- **If construct**

The if construct is used to test the validity of condition.
Syntax of if constructs

```
If (<condition>)
{
        Statement 1;
        statement 2;
        ……..
}
Statement n;
```

This construct evaluates the condition specified in the parentheses and then depending upon whether the value of the condition is true (non-zero) or false (zero) it instructs the program to transfer the control to a particular statement.

If the condition is true, the statements included in the block after the if statement are executed sequentially.
Example

```
/*program to check whether the persons eligible to vote */
#include <stdio.h>
main ( )
{
        int age;
        printf ( " enter the age of voter");
        scanf(" %d", &age);
        if (age >=18)
        {
                printf("\n you are statements to vote \n");
                printf("get your name register today \n");
        }
}
```

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

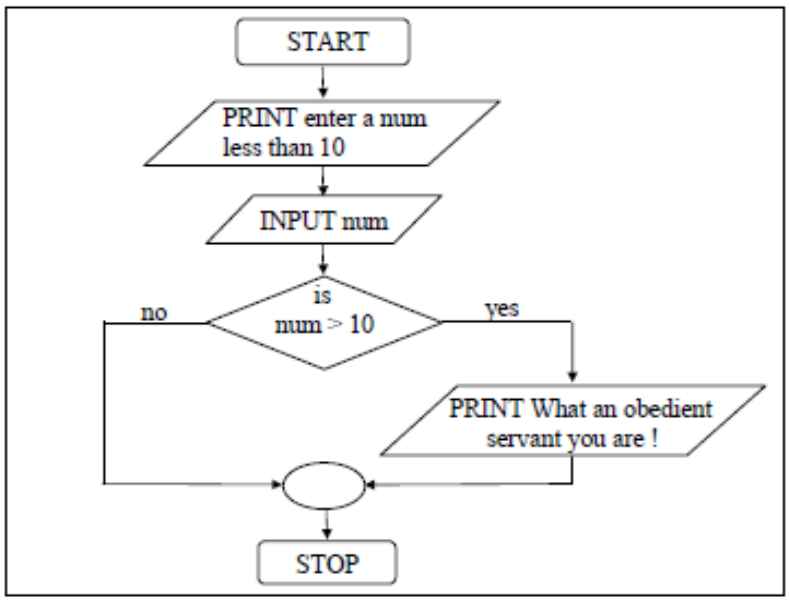
```
if ( this condition is true )
  execute this statement ;
```

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

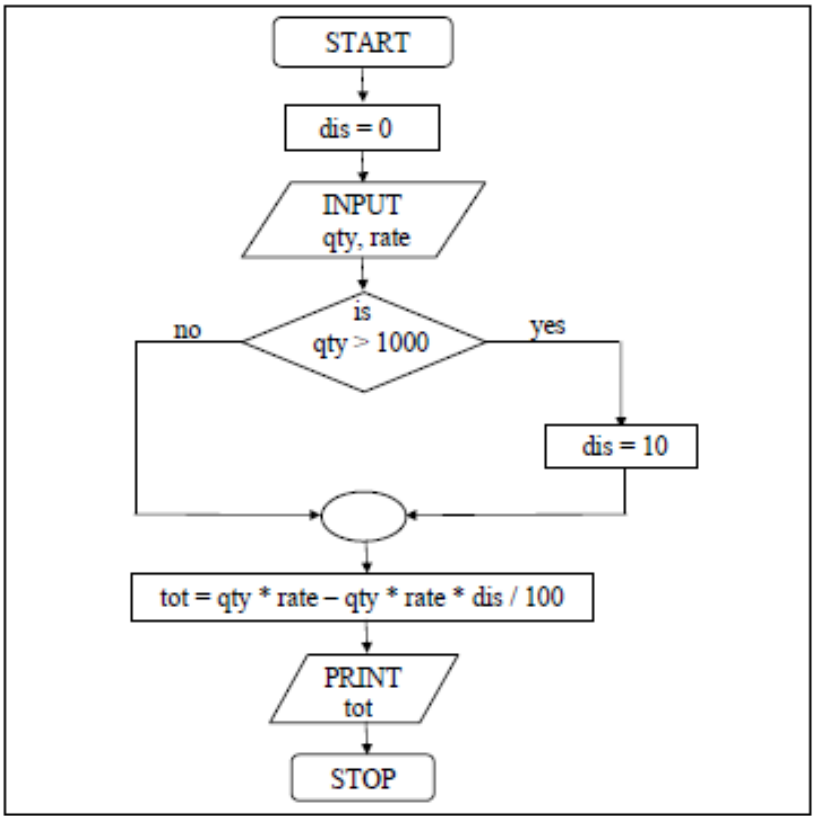
```
/* Demonstration of if statement */
main( )
{
    int  num ;

    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.



Now look at another example.

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.




```
/* 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).

4.2 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.

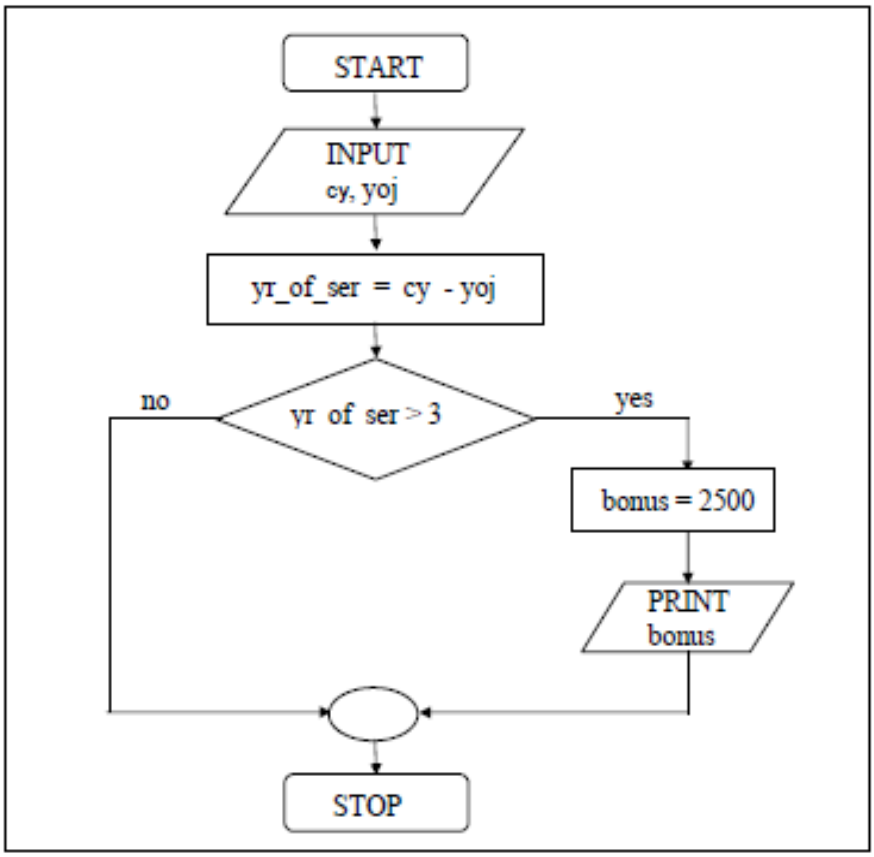
4.3 Multiple Statements within if

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.

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.



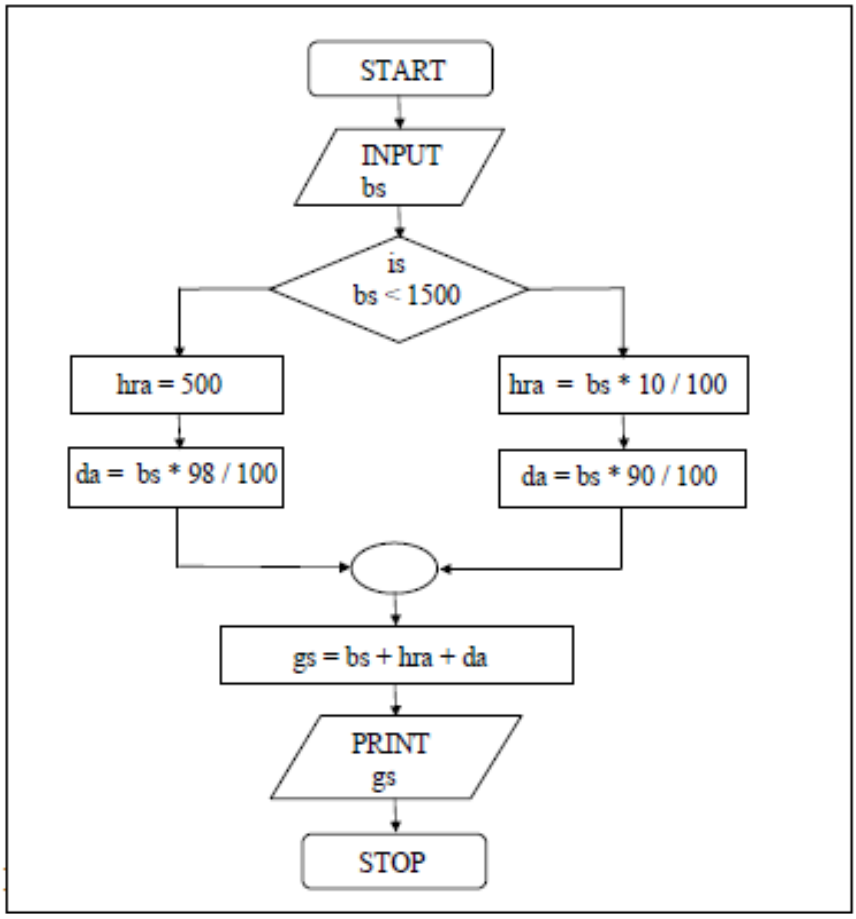
4.4 The if-else Statement

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:

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 ) ;
}
```



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.

4.5 Nested if-else

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 ifs and the elses can be nested.

4.6 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 ;
```

4.7 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. 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.

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:

- 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.
- Care needs to be exercised to match the corresponding ifs and elses.
- 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 ifs with their corresponding elses gets avoided, since there are no elses 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 ifs with their corresponding elses and matching of opening and closing braces that much more difficult.

4.8 The else if Clause

There is one more way in which we can write the last program. 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:
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 ifs and elses 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:

- Driver is married.
- Driver is an unmarried male above 30 years of age.
- 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' &amp;&amp; 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).

Loop Control structures

5.1 While loop
5.2 For loop
5.2.2 Nesting of loops
5.3 Odd loop
5.4 Do-while,
5.5 Other statements :
5.5.1 break
5.5.2 continue
5.6 Case Control Structure
5.6.1 Using Switch
5.6.2 Switch Versus If-else ladder
5.6.3 goto

Loop Control structures

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:

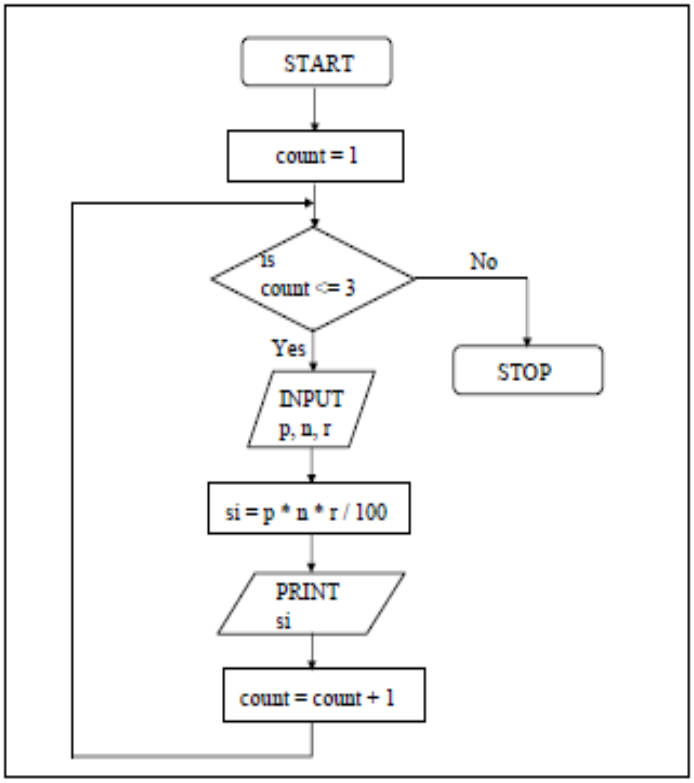
(a) Using a for statement
(b) Using a while statement
(c) Using a do-while statement

5.1 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.

```
/* 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 ;
    }
}
```



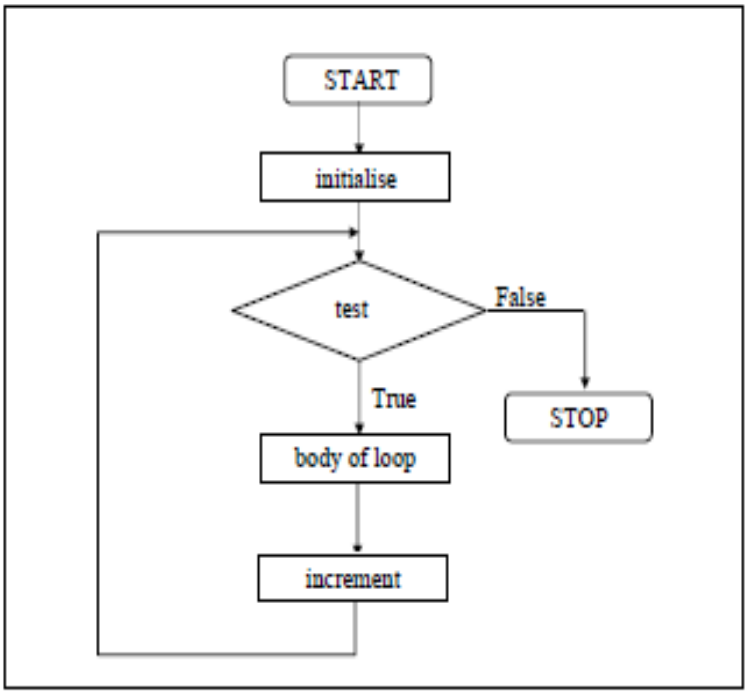
And here are a few sample runs...

Enter values of p, n and r 1000 5 13.5
Sim e 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.



5.2 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:

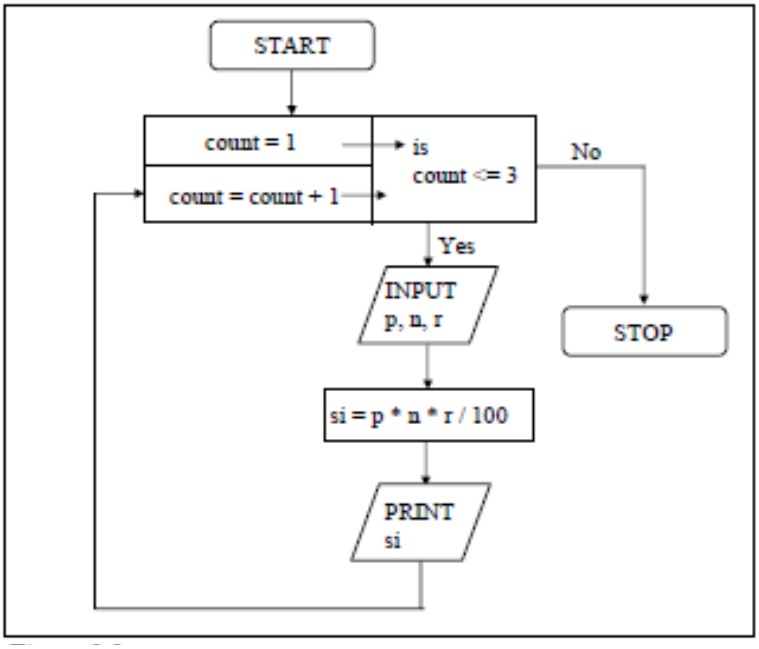
(a) Setting a loop counter to an initial value.
(b) Testing the loop counter to determine whether its value has reached the number of repetitions desired.
(c) Increasing the value of loop counter each time the program segment within the loop has been executed.

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.




```
/* 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.

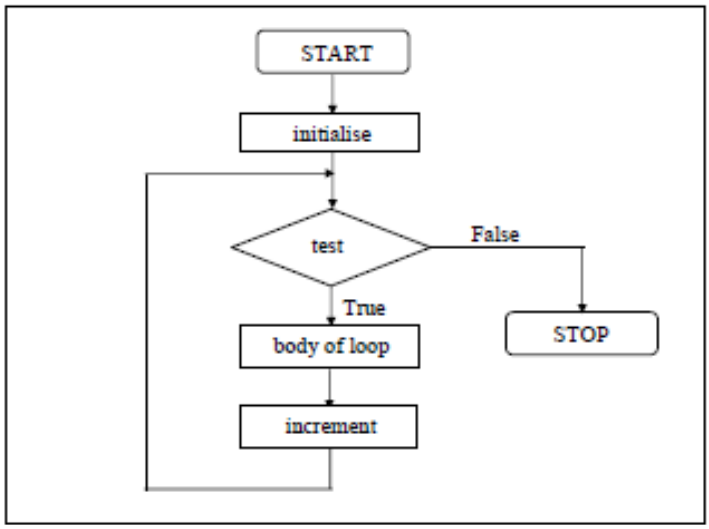
Let us now examine how the for statement gets executed:

- 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.



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 arenecessary.

```
(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.

5.2.2 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.

5.3 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 ) ;
    }
}
```

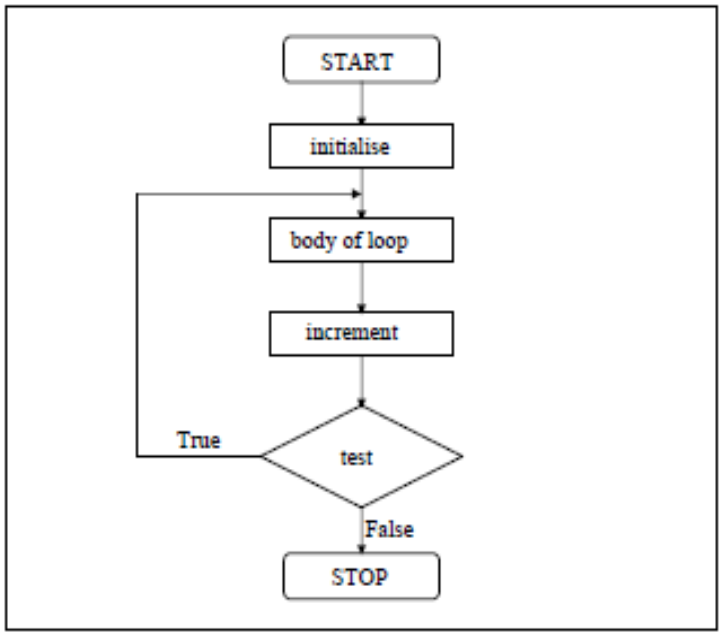
5.4 Do-while

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.



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.

5.5 Other statements :

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. 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.

5.5.1 break

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.

5.5.2 continue

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).

5.6 Case Control Structure

C provides a special control statement that allows us to use rather than using a series of **if** statements.

5.6.1 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

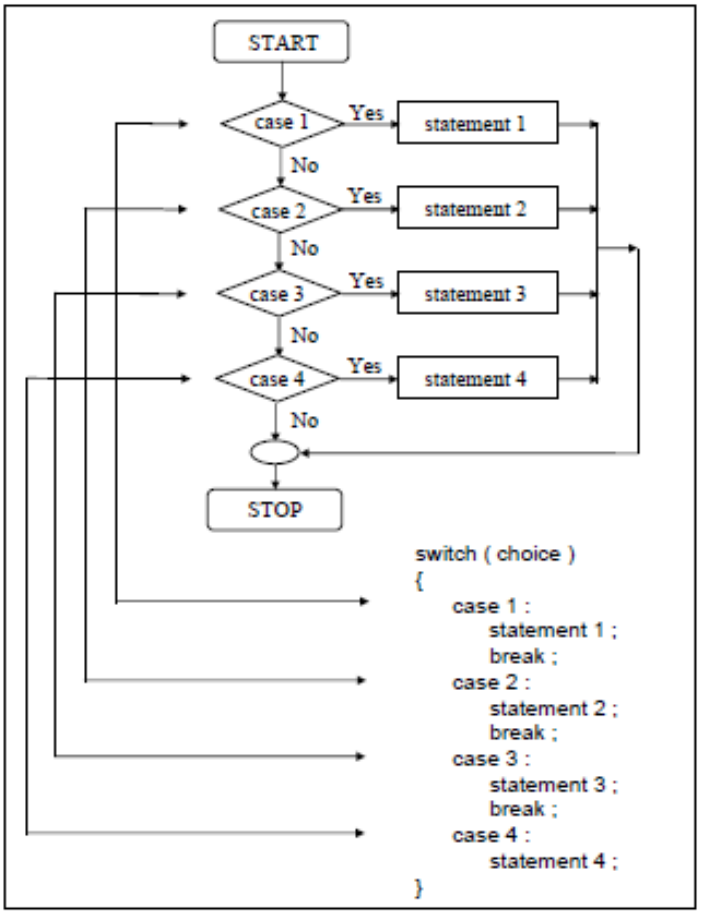
is no need for a break statement after the default, since the control comes out of the switch anyway.

```
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.



5.6.2 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 10thcase 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.

5.6.3 goto

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 gotos 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 ;
e    lse
                    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**

Introduction to problem solving

6.1 Concept: problem solving

6.2 Problem solving techniques (Trial & Error, Brain storming, Divide & Conquer)

6.3 Steps in problem solving (Define Problem, Analyze Problem, Explore Solution)

6.4 Algorithms and Flowcharts (Definitions, Symbols)

6.5 Characteristics of an algorithm

6.6 Conditionals in pseudo-code

6.7 Loops in pseudo code

Introduction to problem solving

6.1 Concept: problem solving

Often just sitting there and being terrified or miffed by the problem will not help solve it. Also, often, no amounting of thinking will solve the problem. So you have to get your hands dirty and grapple with the problem head on. Problem solving is a mental process and is part of the larger problem process that includes problem finding and problem shaping. Considered the most complex of all intellectual functions. Algorithms have come to be recognized as the cornerstone of computing. Algorithm design strategies are typically organized either by application area or by design technique. This report describes different designing algorithms such as Brute force, Greedy, Divide and Conquer, Dynamic programming, Backtracking, Branch and Bound and many more. It describes how a particular algorithm is used for a specific problem. This report also proposes how to choose the best algorithm design strategy for a particular problem to facilitate the development of best algorithms based upon algorithm design strategy techniques. It also describes how a particular algorithm is used for a specific problem. Taking various parameters does a comparison of various algorithms. This report advocates a wider use of different problems in teaching the best algorithm design strategies.

6.2 Problem solving techniques (Trial & Error, Brain storming, Divide & Conquer)

Trial & Error

One of the most tried and true methods for problem solving is the **trial and error** method. Despite its lack of sophistication, sometimes it's the most efficient choice, especially because it can succeed where other methods fail. With trial and error, you're always guaranteed a learning experience, and you'll often identify multiple solutions as you experiment.

Yet how often do we fail to proceed to the trial phase because we're afraid of experiencing the error? We make the mistake of believing that the error is somehow harmful to us, when it is actually helpful. Each error is the feedback we need to formulate a new trial, one that will hopefully lead to new errors and new trials until we ultimately converge on an acceptable solution. So an error is not a failure. An error is in fact merely a step on the path to a success. No errors usually means no successes.

The irony is that we may fear success even more than failure because to our egos, success is a double-edged sword. On the one hand, if your experiment works, then you get the results you want. But on the other hand, if you get the results you want, then you also have to face the fact that you were making a continuous mistake all the while you were "not testing."

For example, if you ask your boss for a raise, and s/he says yes, then the good news is that you get the raise now. But the bad news is that you have to accept that maybe you could have gotten it last year (or sooner) just by asking as you did today. You have to face the fact that fear held you back.

In the long run, it's better to face that fear and to finally achieve the results you want than it is to deny your past fear and simultaneously prevent yourself from making the attempt today. I suspect that in some area of your life, there's a past fear that's still holding you back — an area where you could achieve greater success if you could only allow yourself to be OK with facing up to a previous mistake.

Brain storming

Brainstorming combines a relaxed, informal approach to problem-solving with lateral thinking. It asks that people come up with ideas and thoughts that can at first seem to be a bit crazy. The idea here is that some of these ideas can be crafted into original, creative solutions to the problem you're trying to solve, while others can spark still more ideas. This approach aims to get people unstuck, by "jolting" them out of their normal ways of thinking.

During brainstorming sessions there should therefore be no criticism of ideas: You are trying to open up possibilities and break down wrong assumptions about the limits of the problem. Judgments and analysis at this stage stunt idea generation. Ideas should only be evaluated at the end of the brainstorming session – this is the time to explore solutions further using conventional approaches.

Divide & Conquer

Divide and conquer algorithm suggests splitting the inputs into distinct subsets. These
sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide and conquer strategy can be possibly be reapplied. Often the sub problems resulting from a divide and conquer design are of the same type as the original problem. For those cases the reapplication of the divide and conquer principle is naturally expressed by a recursive algorithm. This algorithm technique is the basis of efficient algorithms for all kinds of the problems, such as quick sort, merge sort and discrete Fourier transform.

6.3 Steps in problem solving (Define Problem, Analyze Problem, Explore Solution)

In order to correctly solve a problem, it is important to follow a series of steps. Many researchers refer to this as the **problem-solving cycle**, which includes developing strategies and organizing knowledge. While this cycle is portrayed sequentially, people rarely follow a rigid series of steps to find a solution. People often skip steps, or even go back through steps multiple times until the desired solution is reached.

Define Problem

Identifying the Problem: While it may seem like an obvious step, identifying the problem is not always as simple as it sounds. In some cases, people might mistakenly identify the wrong source of a problem, which will make attempts to solve it inefficient or even useless.

Defining the Problem: After the problem has been identified, it is important to fully define the problem so that it can be solved.

Analyze Problem

Forming a Strategy: The next step is to develop a strategy to solve the problem. The approach used will vary depending upon the situation and the unique preferences of the individual.

Organizing Information: Before coming up with a solution, we need to first organize the available information. What do we know about the problem? What do we *not* know? The more information that is available, the better prepared we will be to come up with an accurate solution.

Explore Solution

Allocating Resources: Of course, we don't always have unlimited money, time and other resources to solve a problem. Before you begin to solve a problem, you need to determine how high priority it is. If it is an important problem, it is probably worth allocating more resources to solving it. If, however, it is a fairly unimportant problem, then you do not want to spend too much of your available resources into coming up with a solution.

Monitoring Progress: Effective problem-solvers tend to monitor their progress as they work towards a solution. If they are not making good progress toward reaching their goal, they will reevaluate their approach or look for new strategies.

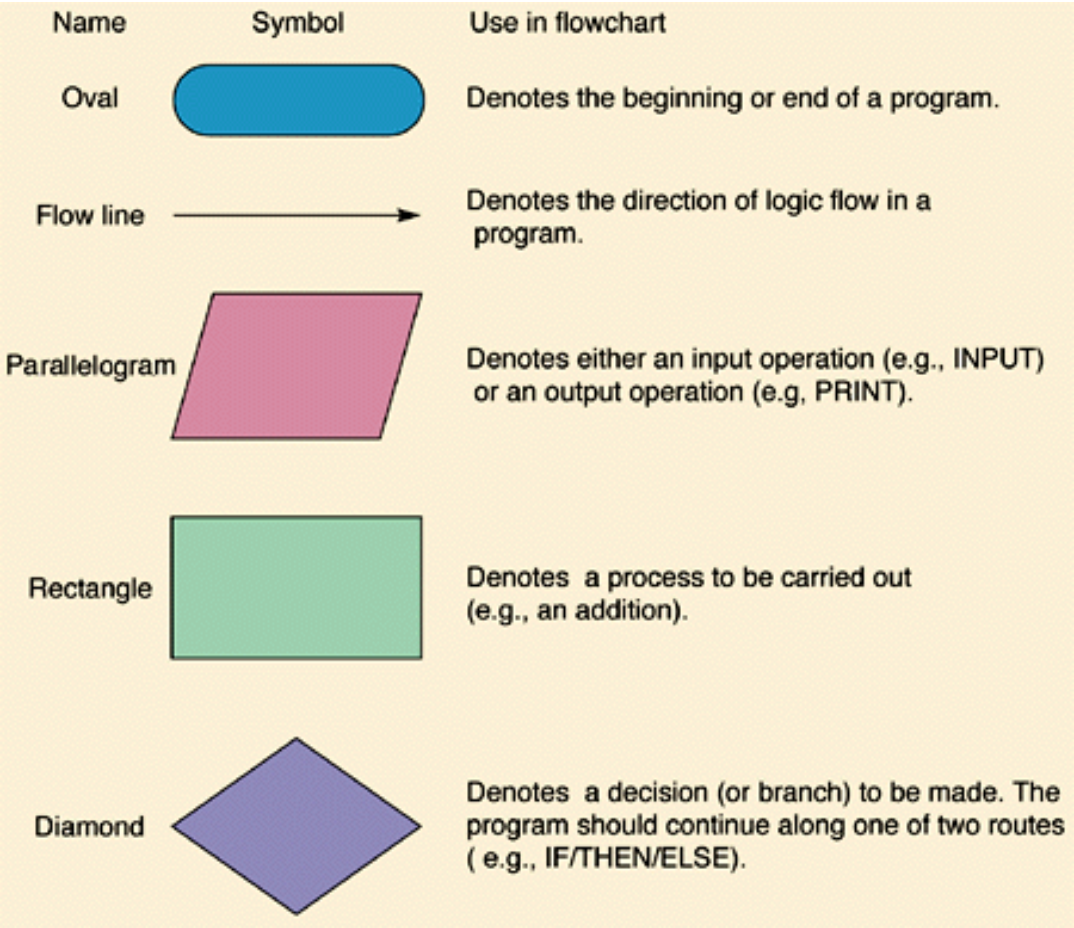
Evaluating the Results: After a solution has been reached, it is important to evaluate the results to determine if it is the best possible solution to the problem. This evaluation might be immediate, such as checking the results of a math problem to ensure the answer is correct, or it can be delayed, such as evaluating the success of a therapy program after several months of treatment.

6.4 Algorithms and Flowcharts (Definitions, Symbols)

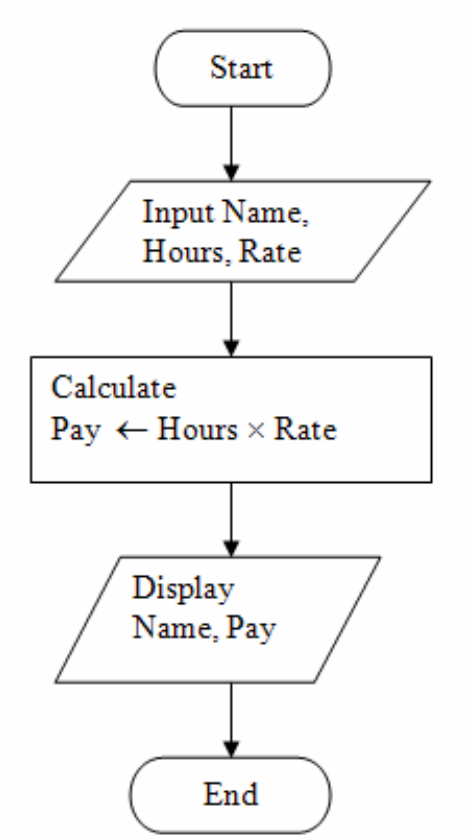
An algorithm is defined as a step-by-step sequence of instructions that describes how the data are to be processed to produce the desired outputs. In essence, an algorithm answers the question: "What method will you use to solve the problem?"

You can describe an algorithm by using flowchart symbols. By that way, you obtain a flowchart which is an outline of the basic structure or logic of the program.

Flowchart: A graphic representation of an algorithm, often used in the design phase of programming to work out the logical flow of a program.



You also can use English-like phases to describe an algorithm. In this case, the description is called pseudocode. Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode has some ways to represent sequence, decision and repetition in algorithms. A carefully prepared pseudocode can be converted easily to a corresponding C++ program.



Flowchart for calculating the payment

End Display Name, Pay Example: The following set of instructions forms a detailed algorithm in pseudocode for calculating the payment of person.
Input the three values into the variables Name, Hours, Rate.
Calculate Pay = Hours * Rate.
Display Name and Pay.

6.5 Characteristics of an algorithm

Every algorithm should have the following five characteristics:

- Input
- Output
- Definiteness
- Effectiveness
- Termination

Therefore, an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input. In other words, viewed

little more formally, an algorithm is a step by step formalization of a mapping function to map input set onto an output set. The problem of writing down the correct algorithm for the above problem of brushing the teeth is left to the reader. For the purpose of clarity in understanding, let us consider the following examples.

Example 1:

Problem : finding the largest value among n>=1 numbers.

Input : the value of n and n numbers

Output : the largest value

Steps :

1. Let the value of the first be the largest value denoted by BIG
2. Let R denote the number of remaining numbers. R=n-1
3. If R != 0 then it is implied that the list is still not exhausted. Therefore look the next number called NEW.
4. Now R becomes R-1
5. If NEW is greater than BIG then replace BIG by the value of NEW
6. Repeat steps 3 to 5 until R becomes zero.
7. Print BIG
8. Stop

End of algorithm

6.6 Conditionals in pseudo-code

The structuring of a program's logic is a fairly recent concept which has been gaining popularity and respect. Structured programming provides the programmer logical constructs from which to build efficient programs. It facilitates simplified logic, easier debugging, and better maintenance. Structured programming is best demonstrated through the use of pseudocode. Pseudocode is an English like language that allows the description of logic to be written, free from the syntax of a formal computer language. Therefore, a particular set of pseudocode becomes portable. That is, it may be shared with different programmers, each of which may work with a different language, and it will be readily understood. Actual computer programming simply becomes the translation of pseudocode into a computer language. The following are four basic control constructs that we will use in pseudocode to give our programs good structure:

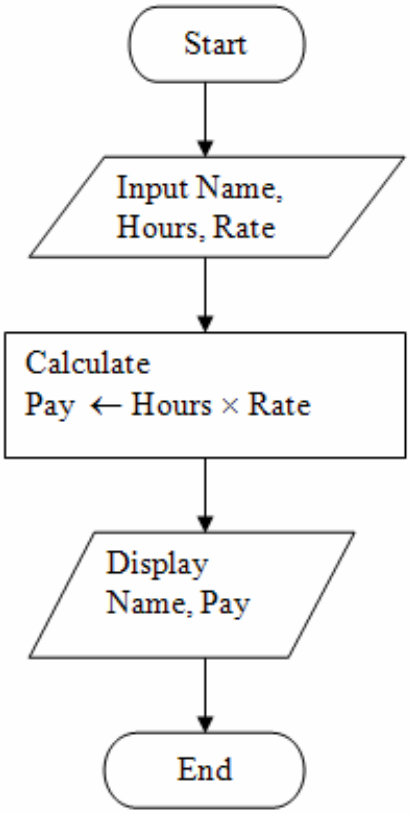
- sequence (straight-line)
- if-else

- if-elseif-else
- do-while

All the above has been explained in the earlier units.

6.7 Loops in pseudo code

You also can use English-like phases to describe an algorithm. In this case, the description is called pseudocode. Pseudocode is an artificial and informal language that helps programmers develops algorithms. Pseudocode has some ways to represent sequence, decision and repetition in algorithms. A carefully prepared pseudocode can be converted easily to a corresponding C program.



Flowchart for calculating the payment

EndDisplayName, PayExample: The following set of instructions forms a detailed algorithm in pseudocode for calculating the payment of person.
Input the three values into the variables Name, Hours, Rate.
Calculate Pay = Hours * Rate.
Display Name and Pay.

Simple Arithmetic Problems

7.1 Addition / Multiplication of integers

7.2 Determining if a number is +ve / -ve / even / odd

7.3 Maximum of 2 numbers, 3 numbers

7.4 Sum of first n numbers, given n numbers

7.5 Integer division, Digit reversing, Table generation for n

7.6 Prime number, Factors of a number 15

7.7 Other problems such as Perfect number, GCD of 2 numbers etc

Simple Arithmetic Problems

By now you will be able to understand the following programs easily. Please perform them as lab exercise also.

7.1 Addition / Multiplication of integers

Write a C program, which takes two integer operands and one operator form the user, performs the operation and then prints the result. (Consider the operators +,-,*, /, % and use Switch Statement)

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

void main()
{
int a,b,res,ch;
clrscr();
printf("\t   ********************");
printf("\n\tMENU\n");
printf("\t*******************");
printf("\n\t(1)ADDITION");
printf("\n\t(2)SUBTRACTION");
printf("\n\t(3)MULTIPLICATION");
printf("\n\t(4)DIVISION");
printf("\n\t(5)REMAINDER");
printf("\n\t(0)EXIT");
printf("\n\t*******************");
printf("\n\n\tEnter your choice:");
scanf("%d",&ch);

if(ch<=5 & ch>0)
{
printf("Enter two numbers:\n");
scanf("%d%d",&a,&b);
}
```

```
switch(ch)
{
case 1:
res=a+b;
printf("\n Addition:%d",res);
break;

case 2:
res=a-b;
printf("\n Subtraction:%d",res);
break;

case 3:
res=a*b;
printf("\n Multiplication:%d",res);
break;

case 4:
res=a/b;
printf("\n Division:%d",res);
break;

case 5:
res=a%b;
printf("\n Remainder:%d",res);
break;

case 0:
printf("\n Choice Terminated");
exit();
break;

default:
printf("\n Invalid Choice");
}
getch();
}
```

7.2 Determining if a number is +ve / -ve / even / odd

Write a program to check whether a give number is even or odd?

```
#include <stdio.h>

int main(void)
{
```

```
int n ;
printf("enter a number :");
scanf("%d",&n);

if(n%2==0)
{
printf("no is even\n");
}
else
printf("no. is odd\n");

return 0;
}
```

Write a program to check whether a give number is positive, negative or zero?

```
#include(stdio.h) // place '><' in place of '(' & ')'
#include(conio.h)
int main( )
{
int no;
clrscr( );
printf("Enter the required number to check:");
scanf("%d", &no);
if(no>0)
printf("\n %d is positive number",no);
else if(no<0)
printf("\n%d is negative number",no);
else
printf("\nEntered Number is ZERO");
getch( );
return 0;
}
```

7.3 Maximum of 2 numbers, 3 numbers

Write a program to find the greatest of three numbers.

```
#include<stdio.h>
int main() {
	int a, b, c;
	printf("Enter a,b,c: \n");
	scanf("%d %d %d", &a, &b, &c);
	if (a > b && a > c) {
		printf("a is Greater than b and c");
	} else if (b > a && b > c) {
```

```
        printf("b is Greater than a and c");
    } else if (c > a && c > b) {
        printf("c is Greater than a and b");
    } else {
        printf("all are equal or any two values are equal");
    }
    return 0;
}
```

Similarly you can write a program to check largest among the two numbers.

7.4 Sum of first n numbers, given n numbers

Write a program to find sum of n numbers

```
#include<stdio.h>
#include<conio.h>
int main(void)
{
int sum=0,n=0;
printf("\n how many numbers do u want to add);
scanf("%d",&n);
for(int j=0;j<n;j++)
{
printf("\n Enter a number");
scanf("%d",&i);
sum=sum+i;
}
printf("\nSum=%d",sum);
return 0;
}
```

7.5 Integer division, Digit reversing, Table generation for n,

Write a c program to accept 3 digits integer reverse it?

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,a,b;
long int revnum=0;
clrscr();
printf("\n Enter three digit number");
scanf("%d", &n);
```

```
a=n%10;
n=n/10;
revnum=revnum+a*100;
a=n%10;
n=n/10;
revnum=revnum+a*10;
a=n%10;
revnum=revnum+a;
printf("\nThe reversed number is %ld",revnum);
getch();
}
```

Write a program to display the multiplication table for a given integer up to 10 using a 'for' loop?

```
#include

void main()

{
int a,no;

clrscr();

printf("Enter a Number that is to be Multiplied\n");

scanf("%d", &no);


for (a=1;a<=10;a++)

{

printf(%dx%d=%d\n", no,a,a*no);

}
}
```

7.6 Prime number, Factors of a number 15

Write a program to check whether the number entered is prime or not?

```
void main()
{
int i,n;
printf("enter the number to be checked");
scanf("%d",&n);
for(i=2;i<=n;i++)
   {
     if(n%i==0)
     printf("The number is prime number")
}
       else
   printf("the given number is not a prime number");
getch();
}
```

Write a C program to find the factors of a given integer.

```
#include(stdio.h>
#include(conio.h)
void main( )
{
int no,x;
clrscr( );
printf("Enter the required number:");
scanf("%d",&no);
printf("\nThe factors are:");
for(x=1; x<=no; x++)
{
if(no%x==0)
printf("\n%d",x);
}
getch( );
}
```

7.7 Other problems such as Perfect number, GCD of 2 numbers etc

Write C programs **that use both recursive and non-recursive functions. To find the GCD (greatest common divisor) of two given integers.**

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

unsigned int GcdRecursive(unsigned m, unsigned n);
unsigned int GcdNonRecursive(unsigned p,unsigned q);

int main(void)
{
int a,b,iGcd;
clrscr();

printf("Enter the two numbers whose GCD is to be found: ");
scanf("%d%d",&a,&b);

printf("GCD of %d and %d Using Recursive Function is %d\n",a,b,GcdRecursive(a,b));
printf("GCD of %d and %d Using Non-Recursive Function is
%d\n",a,b,GcdNonRecursive(a,b));

getch();
}

/* Recursive Function*/
unsigned int GcdRecursive(unsigned m, unsigned n)
{
if(n>m)
return GcdRecursive(n,m);
if(n==0)
return m;
else
return GcdRecursive(n,m%n);
}

/* Non-Recursive Function*/
unsigned int GcdNonRecursive(unsigned p,unsigned q)
{
unsigned remainder;
remainder = p-(p/q*q);

if(remainder==0)
return q;
else
GcdRecursive(q,remainder);
}
```

Exercise:

Write a program;

1. To add n integers and print the result; take the value of n as user input.
2. To print the larger of two numbers.
3. To calculate the area and circumference of a circle, given the radius.
4. That converts Centigrade to Fahrenheit.

   F = (9/5) C + 32.
5. That takes hours and minutes as input, and then outputs the total number of minutes.
6. That takes an integer as the number of minutes, and outputs the total hours and minutes.
7. To tell if the given year is a leap year. [Hint: A leap year is any year divisible by 4, unless the year is divisible by 100, but not 400.]
8. To find the average of n numbers; take the value of n as user input.
9. To print Fibonacci series 0 1 1 2 3 5 8.
10. To find sum of digits of a given number.

Functions

8.1 Basic types of function

8.2 Declaration and definition

8.3 Function call

8.4 Types of function

8.5 Parameter passing
8.5.1 Call by value
8.5.2 Call by reference

8.6 Introduction to Pointers

8.7 Pointer Notation

8.8 Recursion.

Functions

A function is a part of program statements, aimed at achieving a desired result. A function could be written to calculate the larger of two numbers, print a desired message on the screen or calculate the interest payable, etc.

In 'C', as stated earlier, there are 32 key words only; rest of the working of 'C' is handled using function.

8.1 Basic types of function

'C' provides its programmers two types of functions.

1. In-built functions
2. User-Defined function(UDF's)

We have already used two in-built function of 'C', the printf() function and the scanf() function for formatted input and formatted output operations. In the due course of this book, we wick study the other relevant in-built functions available to a programmer in 'C'.

The main() function used by us in all the programs written earlier is a compulsory function for every 'C' program to execute independently.

A 'C' source code file which does not contain the main() function, cannot be executed independently.

In the chapter we will be studying User Define Functions in 'C'.
The major difference between the in-built functions and user-Defined Functions is that, a programmer has to write a user-
Defined Function, whereas in-built functions are available to a programmer in the standard 'C' libraries. Due to this, the in-built functions are also known as library functions.

- **Advantages of Functions**

The strength of 'C' language lies in the fact that functions are easy to define and use.
Functions facilitate the modular programming technique. Functions can be used to divide a large bulky program into functionally-independently modules or sub-programs that can be tested, debugged and maintained independently.

As discussed earlier, UDF's can be made a part of a library which, in turn, facilitates the usage of that function across various other 'C' programs. This feature, thereby, saves the time that one would have to take to record it for another application. This feature is known as "code-reusability" in general programming jargon.

8.2 Declaration and definition

The function in 'C' sre defined in the following format.
Definition of a 'C' function
Function-name([argument list})
[argument declaration];
{
[local variable declarations];
statement 1;
statement 2;
……
[return([expression])];
}

A function declaration in 'C' consist of the function name followed by a pair of open and close parentheses '()'. The function naming convention is the same as that of naming variables.
The parentheses can contain an optional list of arguments that are passed to the function. When arguments (parameters) are passed to a function they are defined before the body of the function (block of statement enclosed in braces).

The body of the function (executable block of statement) is enclosed with in braces. A function should have at least one executable statement. If no executable statement is present in a function, it is known as a dummy function. There is no limitation to the number of executable statements that can be included in the body of a function.
The body of function can also have an optional declaration for the variables to be used by the function only. These variables are known as local variables.
A function can also have an optional return statement. The return statement can contain an optional expression or value that is to be returned back to the calling function.

- **Categories of function**

A function depending upon whether arguments are passed or not whether a value is returned or not, can be classified into the following three categories.

1. Functions with no arguments and no return value
2. Functions with arguments and no return value
3. Functions with arguments and return value

- **Function with no arguments and no return values**

The function dispwelcome() in the above example does not return a value, but still it is used for a specific task. Such functions are called by stating their names as independent statements ending with a semicolon.
When a function has no arguments, i.e., no parameters are passed to the function, it does not receive any data from the calling function. Similarly, when a function does not return any values, it does not pass any information to the calling function. In such a case, there is no communication between the two functions. In effect, there is no data transfer taking place between the calling function and the called function.
The example above using dispwelcome() function, belongs to this category of functions, where no arguments are passed and there are no return values.

8.3 Function call

A function can be called by simply using the function name in a statement.
Example

```
/* program to display welcome message using function */
#include<stdio.h>
main()
{
printf("welcome to STG in main function \n");
dispwelcome(); /* calling function */
printf("welcome to STG in main function again \n");
}
dispwelcome()/* function declaration */
{                    /* Function starts */
        printf("welcome to STG in called function \n *);
}                    /* Function ends */
```

The output of the above program will be

```
        Welcome to STG in main function
        Welcome to STG in called function
        Welcome to STG in main function again
```

8.4 Types of function

- **Categories of function**

A function depending upon whether arguments are passed or not whether a value is returned or not, can be classified into the following three categories.

1. Functions with no arguments and no return value
2. Functions with arguments and no return value
3. Functions with arguments and return value

- **Function with no arguments and no return values**

The function dispwelcome() in the above example does not return a value, but still it is used for a specific task. Such functions are called by stating their names as independent statements ending with a semicolon.
When a function has no arguments, i.e., no parameters are passed to the function, it does not receive any data from the calling function. Similarly, when a function does not return any values, it does not pass any information to the calling function. In such a case, there is no communication between the two functions. In effect, there is no data transfer taking place between the calling function and the called function. The example above using dispwelcome() function, belongs to this category of functions, where no arguments are passed and there are no return values.

- **Function with arguments and no return value.**

Example

```
        /*program to pass two integer parameters to a function and calculate their sum*/
#include<stdio.h>
main()
{
        int firstno, secondno;
        printf("enter two numbers you want tofind sum:");
        scanf("%d %d", &firstno, &secondno);
        printf("\n \n you entered %d %d \n",firstno, secondno);
        calcsun(firstno, secondno); /* calling function to calculate sum and display result*/
}/*end of main function*/
calcsum(paramtr 1, paramtr 2) /* function declaration with parameters */
imt paramtr 1, paramtr 2: /* declaring of the function */
{ /* start of function */
        int sum; /* local variable of the function*/
        sum+ paramtr 1 + paramtr 2;
        printf("the sum of %d %d id %d \n ", paramtr1, paramtr2, sum);
}
/*end of called function*/
```

The example of calculating sum of two numbers, is an illustrate of a function with can have control over the data that is being passed to the called function. The calling function can validate the data called function.

- **Functions with arguments and return value.**

Example

```
/*program to calculate sum of two number in a function and return the sum */
#include<stdio.h>
main()
{
```

```
        int firstno, secondno, sum;
        printf("enter the two numbers you want to calculate sum of :');
        scanf(" %d %d", &firstno, &secondno);
        sum=calsum(firstno, secondno); /8calling function to calculate sum*/
        printf(" \n \n The sum of %d and %d is %d \n", firstno, secondno, sum);
}
/*end of main*/
calcsum (a, b)
int a, b;
{
        int csum;
        csum=a+b;
        return(csum);
/*returning a value to the calling function */
}
/* end of called function*/
```

In the following example we will rewrite the above program again, with some modifications. Analyze the difference between the two programs.

Example

```
 /*program to calculate sum of two number in a function and return the value */
#include<stdio.h>
main()
{
        int firstno, secondno, sum;
        printf("enter the two numbers you want to calculate sum of :');
        scanf(" %d %d", &firstno, &secondno);
        printf(" \n \n The sum of %d and %d is %d \n", firstno, secondno, calcsum(firstno,
        secondno));
}/*end of main*/
calcsum (a, b)
int a, b;
{
        return(a+b);/*returning a value to the calling function */
}/* end of called function*/
```

The above example shows passing of values to the function and the function returning the value. An independent function should act as a black box that receives a pre-defined form of input and outputs a desired value.

This is important, because, a programmer can use the same function repeatedly. Depending upon the requirements of the calling function, the processed information in the called function may be used in one format or the other. Since information manipulation is wholly dependent on the calling function, a programmer cannot format the information in the called function. Thus it

becomes necessary, to return the value back to the calling function, where necessary processing is carried out.
In this category of function, there is a two way data communication between the calling function and the called function.
The calling function can authenticate the data that is passed to the function and the called function returns the processed data back to the calling function.

8.5 Parameter passing

Passing Values to a function
A calling function can send a "package" of values for the called function to operate on, or to control how the function is to operate. Each value passed to a function in such a manner is known as the argument or parameter
Syntax for passing arguments to as function
Function_name (argument 1, argument 2,……argument n);
The arguments passed to a function are separated by commas.
There is no limitation to the number of arguments that are passed to a user Defined function.

- **Actual and Formal arguments**

The data that is passed by the calling function as arguments is known as actual arguments.
Formal or dummy arguments are the names of the parameter in the called function.
Example

```
#include<stdio.h>
main()
{
int a, b, c;
a=100, b=200, c=300;
value(a, b, c); /*Function call made using actual arguments*/
}
value(p, q, r) /*Function declaration*/
int p, q, r;
{
printf("%d %d %d \n",p, q, r);
}
```

In the above program, main() function sends value to the function value().
The values 100 (value of a) , 200 (value of b) and 300 (value of c) are actual parameters which are represented by formal or dummy arguments to the called function(value{}).
The called function (value{}) assigns values 100 to p, 200 to q and 300 to r. 'p','q'and 'r' are formal or dummy parameters.
The actual and formal arguments should match in number, type and order. The values of the actual arguments are assigned to the formal variables on a one to one basis.
It is important to note that for every argument passed with a function, there should be a corresponding argument declaration after the function is defined.
If the number of arguments passed (actual arguments) is more than the argument declarations in the called function (formal arguments), the excess actual arguments are ignored.

However, if the number of actual arguments is less than the formal arguments, the contents of the unassociated formal arguments remain uncertain.
Mismatch in data types of actual arguments and formal arguments will result in passing of un-intelligible data to the function.
All the variables that are used as actual arguments must be assigned values before a function call is made.
It is not necessary, that the variable- name used for passing the arguments (actual parameters) should be the same for the argument declaration list (formal parameters). We can use different variable- names to pass the arguments and receive the parameters in the function.
When a function call is made, only a copy of the values of the actual arguments is passed into the called function. The processing inside the called function does not have any effect on the variables used in the actual argument list.

Returning Values from a function

A function is primarily defined to modularize a big program. The main task of any function is to accomplish a particular task which is describing in the executable statements in the body of the function.
The advantage of a function is lost, when the processing done in the function is not available to the calling function.
After the processing is completed in a function, the information generated has to be passed back to the calling function, so as to enable the calling function make use of the information.
This process is known as returning a value from the function.
To return a value from a function, 'C' provides the return statement.

- **return statement**

return statement is used to pass the value from is called function .

- **syntax or return statement**

return(value);

The function of return statement is two fold:

1. It indicates the end of the called function (it terminates the called function) and transfer the control back to the called function.
2. It sends back value to the calling function.

Boolean Function

Since Boolean values are represented in 'C' by integers, it is very easy for a programmer to write functions that returns integer for a true or false condition.
A Boolean function is useful when a particular test is used many times, especially when the test is somewhat complex. It can make the program efficient and lot more readable.

Example

```
/*program to test for leap year using Boolean function*/
#include<stdio.h>
main()
{
```

```
        int askyear;
        printf("enter the year you want to test:");
        scanf("%d",&askyear);
        if(leapyear(askyear))/*calling Boolean function*/
                {
                printf("%d is a leap year \n",askyear);
                }
        else
                {
                printf("%d is not a leap year \n", askyear);
                }
}/* end of main*/
leapyear(sentyear)/* called function…acts as Boolean function */
int sentyear;
{
        return(sentyear % 4 == 0 && sentyear % 100 ! =0 || sentyear % 400 ==0);
}
/* End of called function*/
```

8.5.1 Call by value

In 'C', by default all the values passed to a function are passed by value.
The process of passing the values of variables using actual arguments is known as call by value. Call by value passes the values of the actual arguments into temporary formal variables. A called function cannot access the actual memory location of the passed argument and therefore, cannot change the contents.

Example

```
/* program to illustrate call by value */
#include<stdio.h>
main()
{
        int firstno =10;
        printf(" The value of firstno in main function is %d \n", firstno);
        change(firstno); /* calling a function */
        printf(" The value of firstno after change function in main function is %d \n", firstno);
}
change(firstno)
int firstno;
{
        firstno =20;
        printf("The value of firstno in change function is %d \n", firstno);
        return(firstno);
}
```

The output of the above program will be
The value of function in the function is 10.
The value of firstno in change function is 20.
The value of firstno after change funtion in main function is 10.

This example illustrates the call by value feature of 'C'.
The Function change () is not able to modify the value of the parameter passed (firstno), because the parameter is passed as a temporary variable. The actual variable is as it is available to the calling function.
All the processing in the called function is localized to the called function only. It does not affect functioning or values of variable in other functions.

8.5.2 Call by reference

In call by reference, 'C' allows the calling function to access the actual memory location of the passed argument and therefore the called function can changes the value of the argument passed.
When we pass the memory address of the argument to the calling function, the phenomena is known as call by reference.
Call by reference makes use of pointers. Pointers are used to pass the memory address of the argument to the called function.
We will take up call by reference again, when we discuss pointers in 'C'.

Local Variable and Global Variable

- **Local (internal) variables**

Variables declared within functions are called local variables.
These variables are local to the function in which they are defined. They local variables have meaning only in the function where they are defined.
Local variables exist only while the function that contains them is executing.

Example

```
Myfunction()
{
        int firstno, secondno; /*Local integer variabl;es */
        char firstchar, secondchar; /* Local character variables*/
}
```

- **Global (external) Variables**

Global or external variables are not declared within a specified function. After a global variable has been declared, it is available to all the functions of the program. The global variables cannot be defined in the function. They are available to the functions by their definition itself. If local variables are given the same name as the global variables within a function, the global variables will not a available to that function.

Example

```
/*program to illustrate global variable*/
#include<stdio.h>
int firstno, secondno; /*Global declaration*/
char firstchar, secondchar; /* Global declaration */
main()
{
        /*Variables fitsfirstno, secondno, firstchar, secondchar available */
        firstno=10;
        secondno=20;
        firstchar='A';
        secondchar='B';
printf("the value of firstno is %d, secondno is %d, firstchar is %c, secondchar is %c before dummy function \n", firstno, secondno, firstchar, secondchar);
dummy();/*calling function */
printf(" The value of firstno is %d, secondno is %d, firstchar is %c, secondchar is %c after dummy function \n", firstno, secondno, firstchar, secondchar);
}/*End of main function*/

dummy() /* Another function in program*/
{
/* Variables firstno, secondno, firstchar, secondchar aviable */
firstno=100;
secondno=200;
firstchar='Z';
secondchar='Y';
printf(" The value of firstno is %d, secondno is %d, firstchar is %c, secondchar is %c in dummy function \n", firstno, secondno, firstchar, secondchar);
}/*End of dummy function*/
```

The output of above program will be:

The value of firstno is 10, secondno is 20, firstchar is A, secondchar is B before dummy function
The value of firstno is 100, secondno is 200, firstchar is Z, secondchar is Y in dummy function
The value of firstno is 100, secondno is 200, firstchar is Z, secondchar is Y after dummy function

- Scope of variables
  The following figure illustrates the scope of variables in 'C'.

```
Main()
{
        local variable a, b, c
        executable statements
        function1(a, b, c)
}
local variables a, b, c aviable
values of a, b, c cannot be changed by alterations in
function1() etc.
Variables defined in function1() not available
```

```
Function1(a, b, c)/* parameters passed*/
Formal variables a, b, c
{
        local variables d, e, f
        executable statements
        alterations a, b, c
        return
}
formal variables a, b, c aviables ass temporary
variables
local variables d, e, f aviable
```

Figure illustrate scope of local variables in 'C'

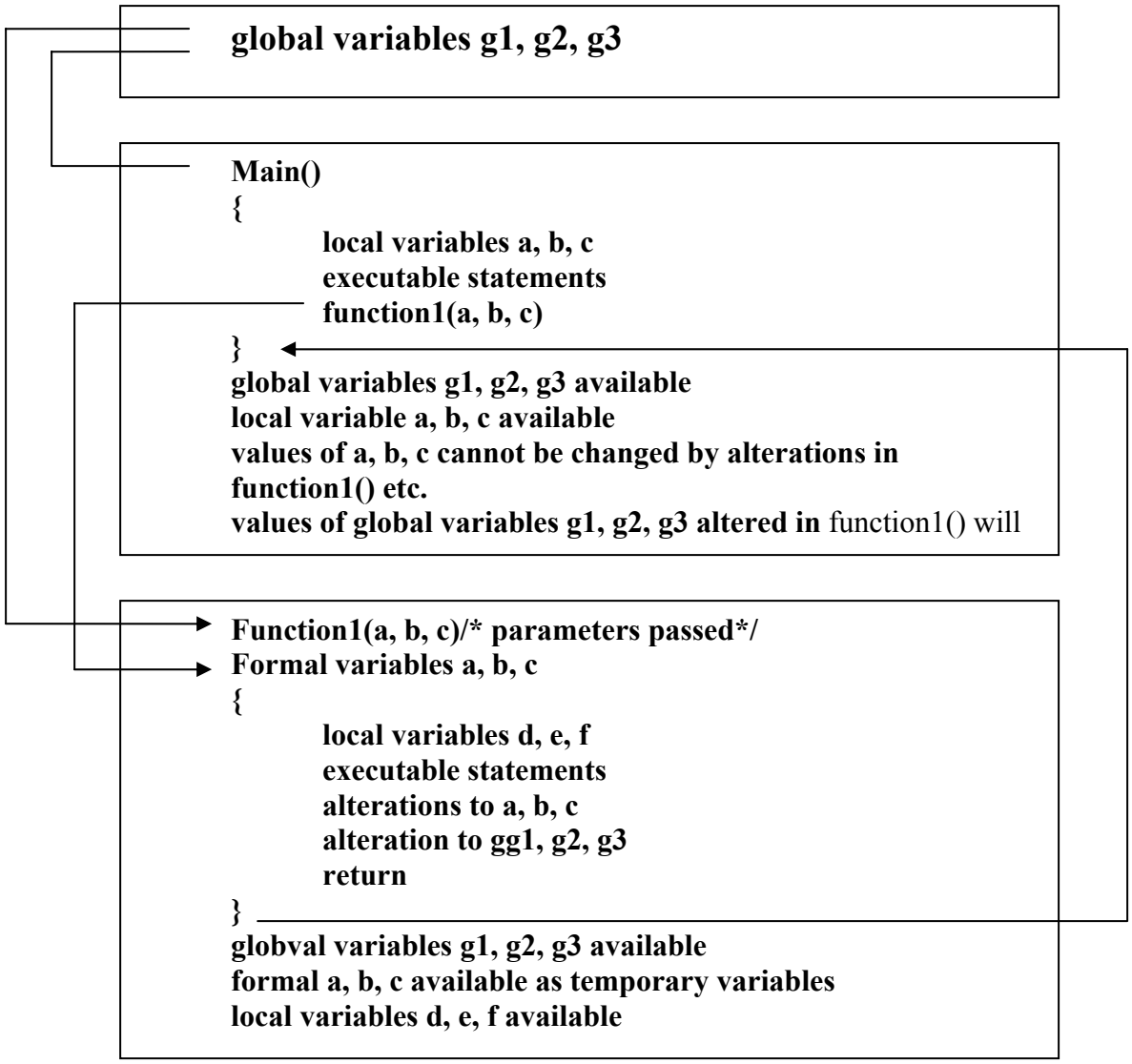


Figure illustrate scope of global variables in 'C'

Functions in ANSI 'C'

Till now, in this chapter we have discussed functions that are for pre-ANSI 'C' or Classical 'C'. Majority of 'C' compilers in the market now-a-days, are based on the ANSI 'C' standards, and functions written in classical 'C' might not be supported on all 'C' compilers.
In this section we will learn about ANSI 'C' functions.

ANSI standards combine the function definition and the arguments declaration in one line (the function header contains both the function name and the data type and name of parameters that are passed, if any).

General form for an ANSI 'C' function:
<type specifier> <name of_the_function>([<type specifier parameter1>, <type specifier parameter2> ……<type specifier parametern>])

If we do not specify the type specifier before the name_of_the_function, by default, a function is treated as an int function.

In case, parameters are passed to a function, the ANSI 'C' standards specify that each of the parameters must be independently declared with their type specifier and name.
ANSI 'C' standards do not permits combinations of more that one argument of same data type by specified only one type specifier alone.

For example.
Int functionone (int a, int b, char choice)
Is a valid function definition as per ANSI "c" standards, and
Int functionone (int a, b, char choice)
Is an invalid function definition.

- **Function Prototype**

As per ANSI standards, every function that is being referenced should be declared for its type in the calling function.
The function definition which includes in the function header, the type of the return value of that function and the data type for all the arguments that is to be passed to the function is known as function prototype,
The introduction of function prototype requires that the function declaration must include not only the type of the value that is to be returned by the function (specified by the type of return value of that function), but also the data type and number of arguments passed to the function. This prototyping as iterated earlier is done in the calling function.

Example

```
/* program to illustrate function prototyping  the program modifies the function used earlier to calculate sum of two numbers as per ANSI C standards The programpasses two integers to a function and calculates their sum in the called function */
#include<stdio.h>
main()
{
        int firstno, secondno;
        void  calcsum(int firstno, int secondno); /* fumction prototype , ANSI standards */
        printf("Enter two numbers to calculate sum:");
        scanf("%d %d", &firstno, &secondno);
        printf("The two numbers are %d and %d \n ", firstno, secondno);
        calcsum(firstno, secondno);/* Calling the function*/
}/* End of main*/
void calcsum(int paramtr1, int paramtr2)
{
```

```
        int sum;
        sum=paramtr1+paramtr2;
        printf("The sum of %d and %d is %d \n", paramtr1, parmtr2, sum);
}/* End of function*/
```

8.6 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.

8.7 Pointer Notation

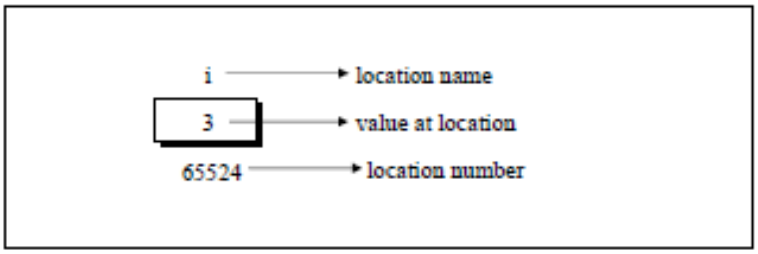
Consider the declaration,

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



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 ) ) ;
}
```

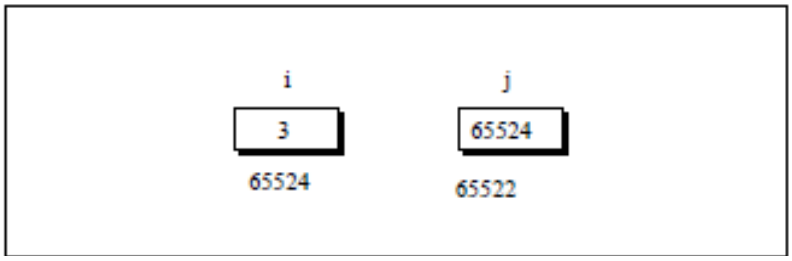
The output of the above program would be:

```
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.



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 thisvariable 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:

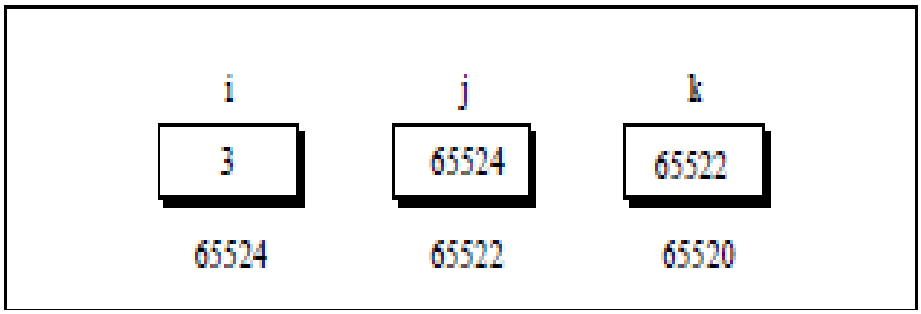
Address of i = 65524

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 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.



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.

8.8 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.

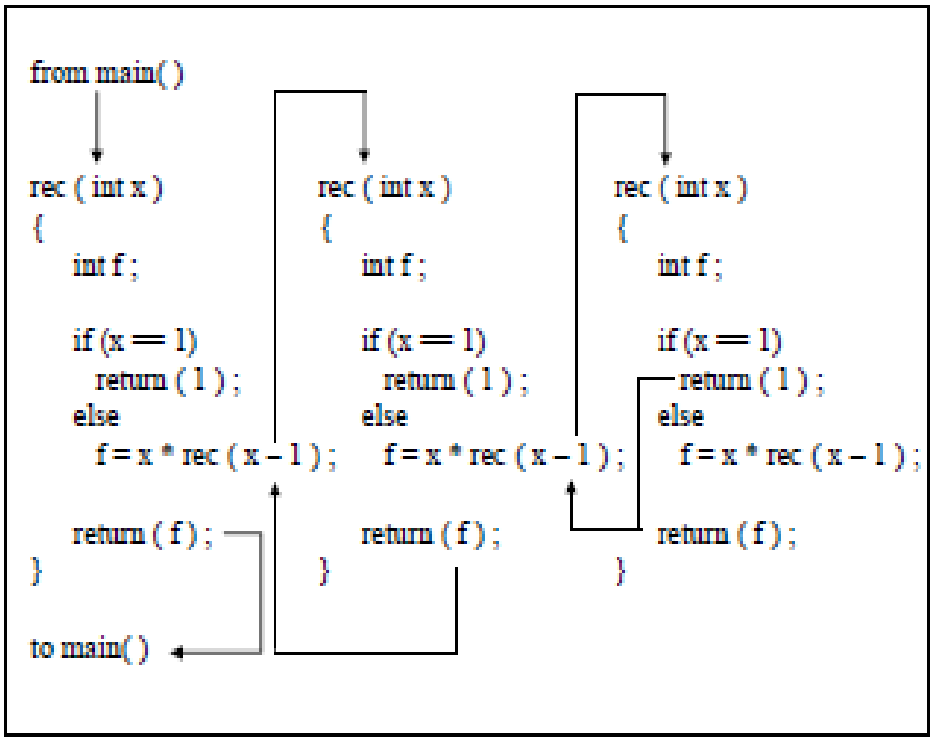
We speak of this series of calls to rec( ) as being different invocations of rec( ). These successive invocations of the same function are possible because the C compiler keeps track of which invocation calls which. These recursive invocations end finally when the last invocation gets an argument value 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. 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.



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.

Storage Class

9.1 Automatic Storage Class

9.2 Register Storage Class

9.3 Static Storage Class

9.4 External Storage Class

Storage Class

We had studied how we can define and use functions in 'C'. In this chapter, we will look into functions again with certain advanced features that 'C' permits with functions.
Also, we will take a look at some of the standard in-built function and the associated standard header files that 'C' provides.
Besides these, we will also look into at the different storages classes available to store data in 'C'.
Storage Classes in 'C'
When we define or declare a variable in 'C', automatically a storage class is associated with the variable.
Storage class define

1. The life span of a variable
2. Availability (or scope) of the variable

The life span of a variable specifies the duration for which the variable will retain a value. This is indirectly associated with the place of declaration and type of variable. If a variable is declared within a function, it will exist as long as the function is executing, but if we have declared a global variable, it will retain the value for the entire program execution period.

Scope of a variable, indicates the parts of the program where the variable is usable. A variable could be available to a block of statements, a function, a program file, a group of program files or the entire application.

With respect to a 'C' compiler, the variable name identities a physical location with the computer, where the variable is to be located. There are basically two type of storage location available inside a computer, viz., the main memory in the central processor unit and the registers. To the compiler, the storage class specifies where the variable is to be stored, in the memory or in the registers.

9.1 Automatic Storage Class

Variable belonging to the automatic storage class are also known as the local variables. This is because, this type of variable are available only to the function in which they are declared. Each local or automatic variable comes into existence in a routine or function only when the function is called, and disappears when the functions is terminated. Automatic variables are dynamic in nature.

Since, automatic variable come and go with function invocation, they do not retain their values from one function call to the next, and must be explicitly set upon entry. This means, that we need to initialize the automatic variable or assign values to them. In case we do not set the value, these variables will contain unknown values.
By default, whenever a available is defined inside a function, it is assigned the automatic storage class.

Optionally, a programmer can use auto to specify to the 'C' compiler that the variable defined is of the automatic class.

Example

```
/* program to illustrate use of automatic variable in C*/
#include<stdio.h>
main()
{
        int firstno=10, secondno=20;
        printf("\n\n In main function, the value of firstno is %4d and the value of second no is
        %4d \n", firstno, secondno);
        function1();/*Activating Function*/
        printf("\n \n In main function again, the value of firstno is %4d and the value of second
        no is %4d \n", firstno, secondno);
        printf("End of main function \n");
}
/*end of main*/
function1()
{
        int firstno=110, secondno =120;
        printf("\n\n In function2, the value of firstno is %4d and the value of second no is %4d
        \n", firstno, secondno);
        function1();/*Activating another Function*/
        printf("\n \n In main function again, the value of firstno is %4d and the value of second
        no is %4d \n", firstno, secondno);
        printf("End of function \n");
}/*end of function 1*/
function2()
int firstno=210, secondno =220;
printf("\n\n In function2, the value of firstno is %4d and the value of second no is %4d \n",
firstno, secondno);
printf("End of function \n");
}
/*end of function 2*/
```

The outputs of above program will we
In main function, the value of firstno is 10 and the value of secondno is 20
In function 1, the value of firstno is 110 and the value of secondno is 120
In function 2, the value of firstno is 210 and the value of secondno is 220

End of function 2
In function 1, the value of firstno is 110 and the value of secondno is 120
End of function 1
In function again, the value of firstno is 10 and the value of secondno is 20
End of main function

9.2 Register Storage Class

In a computer, the variables can either be stored in the memory or in the registers. Variables that are to be stored in the registers are declared using the register keyword before the variable declaration.
Registers are similar to memory locations with the difference that they are a part of the processor itself therefore operations on registers are faster than those relating to memory.
Usually, the variables that are frequently used in a program whose access speed is important criterion of its performance are assigned to register storage class.

A variable declaration with register storage class is a request to the compiler that the variables are to be assigned to a register. The declaration is a request because of the limited number of register available on the machine. The compiler tries to accommodate as many variables as possible to registers, but it may not be able to accommodate all of them due to non-availability of registers. The compiler will automatically convert register variables to non-register variables once all the storage space on the registers is filled up.

Only local variables can be declared to belong to the register storage class.
Because a register variable may be stored in a register of the CPU, it may not have a memory address. This means that we cannot use the '&' to find the address of a register variable.

Non-integer Functions

A function in 'C' by default returns a value of the type int unless specified otherwise. To be able to return a non-integer value from a function the types on the function has to be specified explicitly. This means that we have to specify to the 'C' compiler the data-type for the return value of the function, instead of the default int type.
The type-specifier (data type indicator) is specified along with the function header.
Also, by default the calling function receives the return value from the called function in the int type only. To enable a calling function to receive non-integer value, we must:

1. Provides the type-specifier which will indicates the type of data required, along with the function header.
2. Declare the called function at the start of the body function in a manner similar to variable declaration.

The format to provide the type-specifier with the function headeris:
<type-specifier> <function-name> (<parameter list>)
parameter declaration;

```
{
        function statements;
}
```

Example

```
/* Program to calculate simple interest */
#include<stdio.h>
main()
{
        int time;
        float principal, rate, simple interest();
        printf("\n\n This program calculate the simple interest Usiong a non-integer function\n");
        printf("\n Enter the principal amount\n");
        scanf("%f", &principal);
        fflush(stdin);
        printf("\n enter the rate:");
        scanf("%f",%rate);
        fflush (stdin);
        printf("\n Enter the time:");
        scanf("%d", &time);
        printf("\n The simple interest comes out to be %f \n", simple_interest (principal, rate,
        time));
}/*End of main*/
float simple_interest (principal, rate, time)
float principal, rate;
int time;
.*same variable names used for the passed parameters*/
{
        return ((principal *rate*time)/100);
}/*End of function*/
```

In the above program, we have declared the function simple_interest as float along with the variable declaration. The parentheses '()' after the variable simple_interest name specifies to the 'C' compiler that a function definition has been made.

We can also declare the function as float before main() function declaration. In such a case, we need not specify the type specifier for the function inside the main() declaration.

9.3 Static Storage Class

Static variables are variables that are available in function in which they are defined and they do not loose their value when the function is terminated. This type of variables is present throughout the program but is only accessible in the function in which they were originally defined.
To define static variables, the static reserved word is used.

A static variable may either be an internal static variable or an external static variable, depending on the location of declaration.
Internal static variables are the variables that are defined within a function. It can be only used in the function in which they are declared. Internal static variables can be compared with the automatic variables with respect to the scope of the variables which is the function in which these are defined.
When we use the static modifier, we cause the contents of a local variable to be preserved between calls. Also, unlike normal local variables, which are initialized each time a function is entered a static local variable is initialized only once. For example, take a look at the following program:

Example

```
/*program illustrating internal static variables*/
#include<stdio.h>
main()
{
        int I;
        for(i=0; i<3;i++)
        increment();
} /*End of main*/
increment()
{
        static int ctr=0;
        ctr++;
        printf("ctr=%d \n", ctr);
}/* End of function*/
```

The output of the above program will be
ctr=1
ctr=2
ctr=3

In the above example, the first call to the function increment, declares the variable ctr as static integer and initializes it to 0. Subsequent calls increase the value stored in the variable ctr.
In case, we had declared the variable ctr as automatic or local, the output of the program would have been:
Ctr=1
Ctr=1
Ctr=1

This is because every time the function is called, the variable ctr would have been initialized and the increment would have assigned the value 1 to it.

When a static modifier is used on global variable, it cases the global variable to be known to and accessible by only the function in the same file in which it is declared.

External static variables are global static variable differ from the internal static variable in the manner the internal static variable are available only to the function in the program file in which the variable is defined and not in other program file. Where as, external static variable are available across program file.

Global variable declared as static are useful when a program file contain a group of function that share information with one another by way of global variables, but that information is not needed by function in other program files.

If a global variable is declared as static in one program file, another program file can also declare a global variable with the same name as static without causing any conflict.
When we define a function static by prefixing static before the function name, the function is available to all the function in the programmed and not to function in other program file. A function defined static, is local to the program file in which it was declared.
Neither a global variable nor a function can be declared to be of the auto storage class.

9.4 External Storage Class

Variables that belong to external storage class are available throughout the entire program. These variables are also known as global variables. To define a variable external, the extern reserved word is used before the variable names.
Since all the global variables i.e., the variables declared before all the functions in a program, belong to this class, it is not required to use extern before such variable names. In case, a programmer requires to declare global variables inside a function, then extern is needed.
The extern storage class does not create a variable, but merely informs the compiler of the scope of the variable. An extern declaration can never include an initialization, since the variable is not created when it is declared.
An extern variable has to be assigned values after it is declared. When a value is assigned to an extern (global) variable, the same is available throughout the program. A programmer however can change the value in any other function. The variable, then, will contain the modified value.
External or global variable are also used to link variables between more than one program file. To do so, the global variables (variables outside function), are prefixed with extern.

```
/* program to illustrate global / external variable in C*/
#include<stdio.h>
int firstno, secondno;
main()
{
        firstno=10, secondno=20;
        printf("\n\n In main function, the value of firstno is %4d and the value of second no is
        %4d \n", firstno, secondno);
        function1();/*Activating Function*/
        printf("\n \n In main function again, the value of firstno is %4d and the value of second
        no is %4d \n", firstno, secondno);
        printf("End of main function \n");
}
```

```
/*end of main*/
function1()
{
        firstno=110, secondno =120;
        printf("\n\n In function2, the value of firstno is %4d and the value of second no is %4d
        \n", firstno, secondno);
        function1();/*Activating another Function*/
        printf("\n \n In main function again, the value of firstno is %4d and the value of second
        no is %4d \n", firstno, secondno);
        printf("End of function \n");
}/*end of function 1*/
function2()
{
        firstno=210, secondno =220;
        printf("\n\n In function2, the value of firstno is %4d and the value of second no is %4d
        \n", firstno, secondno);
        printf("End of function \n");
}/*End of function 2*/
```

The outputs of above program will we
In main function, the value of firstno is 10 and the value of secondno is 20
In function 1, the value of firstno is 110 and the value of secondno is 120
In function 2, the value of firstno is 210 and the value of secondno is 220
End of function 2
In function 1, the value of firstno is 210 and the value of secondno is 220
End of function 1
In function again, the value of firstno is 210 and the value of secondno is 220
End of main function

Decision Control structures

4.1 The if Statement
4.2 The Real Thing
4.3 Multiple Statements within if
4.4 The if-else Statement
4.5 Nested if-else
4.6 Forms of if
4.7 Use of Logical Operators
4.8 The else if Clause
4.9 Nested if-else
4.10 Use of Logical Operators
4.11 The else if clause