# CHAPTER 1: INTRODUCTION TO C

## 1.1 INTRODUCTION TO C

### 1.1 .1 INTRODUCTION:

Powerful features, simple syntax, and portability make C a preferred language among programmers for business and industrial applications. Portability means that C programs written for a computer with a particular kind of processor say Intel can be executed on computers with different processors such as Motorola, Sun Sparc, or IBM with little or no modification.

C language is widely used in the development of operating systems. An **O**perating **S**ystem (OS) is software (collection of programs) that controls the various functions of a computer. Also it makes other programs on your computer work. For example, you cannot work with a word processor program, such as Microsoft Word, if there is no operating system installed on your computer. Windows, UNIX, Linux, Solaris, and MacOS are some of the popular operating systems.

### 1.1.2 HISTORY OF C:

In the early days of computers, the only programming languages available to programmers were two “low level” languages, machine and assembly language. Programming in machine and assembly languages was tedious and time consuming because all memory, stack, interrupts and hardware is to be managed by programmer. Increase in use of computers, many “high-level” languages such as FORTRAN, Basic and Pascal are developed.

By using “high-level” languages hardware will be simple, syntax made easier. In 1972, the Unix Operating system was being developed. During this time, the concept of a System programming language having attributes of both “low level” and “high level” languages also developed. Brian W. Kernighan and Dennis M. Ritchie developed C at Bell Laboratories as a system programming language. Their goal was to develop a language that was simple and flexible enough to be used in a variety of different platforms. In 1983, the American National Standards Institute formed a committee to produce a C programming language standard. This "ANSI C" was completed in 1988. C with objects) is the most popular programming language in the world and is the dominant in Microsoft Windows application development. Small C can be used in microcontrollers.

### 1.1.3 CHARACTERISTICS OF ‘C’:

- Modularity
- Portability

- Extendibility
- Speed
- Flexibility

**Modularity:**

Ability to breakdown a large module into manageable sub modules called as modularity, which is an important feature of structured programming languages.

**Advantages:**

1. Projects can be completed in time.
2. Debugging will be easier and faster.

**Portability:**

The ability to port i.e. to install the software in different platform is called portability.

*Highest degree of portability:* 'C' language offers highest degree of portability i.e., percentage of changes to be made to the sources code is at minimum when the software is to be loaded in another platform. Percentage of changes to the source code is minim.um. The software that is 100% portable is also called as platform independent software or architecture neutral software. Eg: Java.

**Extendibility:**

Ability to extend the existing software by adding new features is called as extendibility.

**Speed:**

'C' is also called as middle level language because programs written in 'c' language run at the speeds matching to that of the same programs written in assembly language so 'c' language has both the merits of high level and middle level language and because if this feature it is mainly used in developing system software.

**Flexibility:**

- Key words or reserved words.
- ANSIC has 32 reserved words.
- 'C' language has right number of reverse words which allows the programmers to have complete control on the language.
- 'C' is also called as programmer's language since it allows programmers to induce creativeness into the programmers.

1.1.4 WHAT KIND OF LANGUAGE IS C?

C is called a **middle-level language** because it combines the best elements of low-level or machine language with high-level languages.

- C takes a middle path between low-level assembly language…
  - Direct access to memory layout through pointer manipulation
  - Concise syntax, small set of keywords
- …and a high-level programming language like Java:
  - Block structure
  - Some encapsulation of code, via functions
  - Type checking (pretty weak)

C is a **Structure Programming Language** because it contains programs in well defined control structures. C language can break large task into sub-tasks which made programmer easy called Modularity.

C VS ASSEMBLY:

| **C** | **ASSEMBLY** |
|---|---|
| Writing Programs much faster | Writing programs are slower |
| Learning and mastering takes less time | It takes more time |
| Portable | Non portable |
| Well defined structured | Not a well defined structured |
| Powerful keywords | More number of keywords |
| Supports modularity | Does not support modularity |
| High-level language | Low-level language |

1.1.5 WHY YOU SHOULD LEARN C?

- C is simple.
- There are only 32 *keywords* so C is very easy to master.
- C programs run faster than programs written in most other languages.
- C enables easy communication with computer hardware making it easy to write system programs such as *compilers* and *interpreters*.
- C language supports Integrated Development Environment (IDE).

1.1.6 ROLE OF ‘C’ IN EMBEDDED SYSTEMS:

C is one of the most commonly used software languages used on embedded device controllers. One reason is because it is one of the few software languages that operate on both 8 bit controllers and 64 bit PCs, meaning that many computer programmers can write

C software for both personal computers and embedded devices. The C language can also use very simple commands to control the device, freeing up the limited memory of the device to hold many commands or parameters. C can be written for both microcontrollers and digital signal processors.

Code is written in C on a programmer's PC. Code is run through a compiler on the programmer's PC to create a software program. The embedded system software may be run through a simulator on the programmer's computer. The software program is copied onto the controller using a "programmer." The controller is then tested on a "test bed" to ensure that it works properly.

### 1.1.7 COMMON EMBEDDED SYSTEMS THAT USE C:

Bluetooth devices are programmed in C. PIC microcontrollers such as those used in web cameras are frequently programmed in C. PIC microcontrollers programmed in C have also been used in LED (light emitting diodes) devices and LCD (liquid crystal display) monitors. USB devices are embedded devices frequently coded in C.

### 1.1.8 STANDARDS FOR EMBEDDED SYSTEM PROGRAMMING IN C:

The American National Standards Institute (ANSI) has written standards for the C programming language. The International Standards Organization wrote standard ISO/IEC 9899 for the C programming language. The Motor Industry Software Reliability Association has created a proprietary set of standards for programming in C for embedded devices in automobile.

### 1.1.9 WHERE IS C USEFUL?

C’s ability to communicate directly with hardware makes it a powerful choice for system programmers. In fact, popular operating systems such as Unix and Linux are written entirely in C. Additionally, even compilers and interpreters for other languages such as FORTRAN, PASCAL, and BASIC are written in C. However, C’s scope is not just limited to developing system programs. It is also used to develop any kind of application, including complex business ones.

The following is a partial list of areas where C language is used:

- Embedded Systems
- Systems Programming
- Artificial Intelligence
- Industrial Automation
- Computer Graphics
- Space Research

- Image Processing
- Game Programming

### 1.1.10 HOW TO RUN A SIMPLE ‘C’ PROGRAM:

Steps given below in order to type and run the program in TURBO C.

- Go to the directory where you have installed Turbo C.
- Type TC at the DOS command prompt.
- In the edit window that opens, type the mentioned program above.
- Save the program as hello.c by pressing F2 or Alt + ‘S’.
- Press Alt + ‘C’ or Alt + F9 to compile the program.
- Press Alt + ‘R’ or Ctrl + F9 to execute the program.
- Press Alt + F5 to see the output.

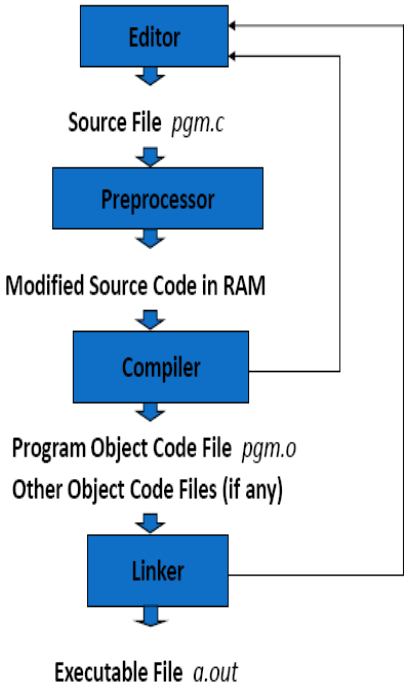
Steps given below in order to type and run the program in LINUX

- Go to the Linux command prompt (# or $)
- Type **vi**
- The **vi** editor will open
- Type the program in the editor
- Press ‘Esc + Shift + ‘:’
- Type ‘**w**’ + ‘**q**’ followed by file name ‘**hello.c**’
- At the command prompt type ‘**gcc hello.c**’
- Type **‘. /a.out**’ to run the program

### 1.1.11 C-COMPILERS:

When you write any program in C language then to run that program you need to compile that program using a C Compiler which converts your program into a language understandable by a computer. This is called machine language (i.e. binary format). So before proceeding, make sure you have C Compiler available at your computer. It comes along with all flavors of UNIX and Linux.

If you are working over UNIX or Linux then you can type *gcc -v* or *cc -v* and check the result. You can ask your system administrator or you can take help from anyone to identify an available C Compiler at your computer.




1.1.12 LIFE CYCLE OF A C PROGRAM:

1) **Write** the program

```
int main ()
{

        printf ("Hello World");
}
```

2) **Compile** the program

When the program gets compiled, the compiler realizes that the current compile unit (ie., the simple C program in this case) has no implementation for printf(), and therefore produces an entry in the object file's symbol table saying that printf() has an `unresolved reference'. And the compiler generates an object file (.o file) if there were no syntax errors in the program.

3) **Link** the object file(s)

The next phase is to link the object files to produce an executable. During linking, the static linker (ld) sees the unresolved reference to printf() and searches the available libraries for an implementation for printf(). In general this will be found in the C library (libC on Solaris). Now, the linker has two options:

It (linker) can take the printf() implementation from the library and copy it into the final executable. The linker then searches the printf() implementation for other unresolved references, and again consult the libraries for resolution. This process will be performed iteratively until all references to the symbols were resolved. This is known as **static linking**

If the C library is realized as a `shared library', the linker can simply put a reference to the C library into the final executable. Still the linker performs symbol resolution checking as above, to determine if the reference to the printf() function necessitates further references to other (shared or nonshared) libraries. This is known as **dynamic linking**

4) **Run** the executable

What happens when you run the executable depends on whether it was linked statically or dynamically:

A statically linked executable is self contained. It is loaded into memory. The entry point, whose designation is system dependent (for eg, the `__main' symbol) is found and called. This entry function, usually provided by the compiler or a library, performs some setup and initialization and then calls the user-defined main() function and the instructions inside main() function gets executed

In a dynamically linked executable, after loading the executable binary into memory, the dynamic linker (ld.so.1) takes control first. It reads the library references to dynamic libraries produced by the static linker, and loads them into memory. It then performs symbol resolution again and updates all references to symbols in the shared library to point to their actual location, which can only be determined at runtime, because the shared libraries might be loaded to different memory locations each time the executable binary gets executed

1.1.13 THE PREPROCESSOR:

A unique feature of c language is the preprocessor. A program can use the tools provided by preprocessor to make his program easy to read, modify, portable and more efficient. Preprocessor is a program that processes the code before it passes through the compiler. It operates under the control of preprocessor command lines and directives. Preprocessor directives are placed in the source program before the main line before the source code passes through the compiler it is examined by the preprocessor for any preprocessor directives. If there is any appropriate actions are taken then the source program is handed over to the compiler. Preprocessor directives follow the special syntax rules and begin with the symbol #bin column1 and do not require any semicolon at the end. A set of commonly used preprocessor directives

Preprocessor directives:

| Directive | Function |
|---|---|
| #define | Defines a macro substitution |
| #undef | Undefines a macro |
| #include | Specifies a file to be included |

| | |
|---|---|
| #ifdef | Tests for macro definition |
| #endif | Specifies the end of #if |
| #ifndef | Tests whether the macro is not def |
| #if | Tests a compile time condition |
| #else | Specifies alternatives when # if test fails |

The preprocessor directives can be divided into three categories

- Macro substitution division
- File inclusion division
- Compiler control division

1.1.14 MACROS:

Macro substitution is a process where an identifier in a program is replaced by a pre defined string composed of one or more tokens we can use the #define statement for the task.

It has the following form

```
#define identifier string
```

The preprocessor replaces every occurrence of the identifier int the source code by a string. The definition should start with the keyword #define and should follow on identifier and a string with at least one blank space between them. The string may be any text and identifier must be a valid c name. There are different forms of macro substitution. The most common form is

- Simple macro substitution
- Argument macro substitution
- Nested macro substitution

Simple macro substitution:

Simple string replacement is commonly used to define constants example:

```
#define pi 3.1415926
```

Writing macro definition in capitals is a convention not a rule a macro definition can include more than a simple constant value it can include expressions as well. Following are valid examples:

```
#define AREA 12.36
```

Macros as arguments:

The preprocessor permits us to define more complex and more useful form of replacements it takes the following form.

```
# define identifier(f1,f2,f3…..fn) string.
```

Notice that there is no space between identifier and left parentheses and the identifier f1,f2,f3 …. Fn is analogous to formal arguments in a function definition.
There is a basic difference between simple replacement discussed above and replacement of macro arguments is known as a macro call
A simple example of a macro with arguments is

```
# define CUBE (x) (x*x*x)
```

If the following statements appears later in the program,

```
Volume = CUBE(side);
```

The preprocessor would expand the statement to

```
volume =(side*side*side)
```

This directive is also called as Macro substitution directive.

Syntax:

**#define <Macro_constant> [(<Para1>, <Para2>,...)] <Token_string>**

Note: [] indicates optional term.

Task of macro substitution directive is to replace the identifier with corresponding Token_string. For example:

```
#include<stdio.h>
#define pie 3.14
int main() {
float r=3,area;
area = 3 * r * pie;
printf("%f",area);
return 0;
}
```

In the above c code we have defined a macro constant pie. Before the starting of actual compilation an intermediate is formed which is:

```
#include<stdio.h>
int main() {
float r=3,area;
area = 3 *r * 3.14;
printf("%f",area);
return 0;
}
```

We can see, only in place of macro constant pie corresponding token string i.e. 3.14 has pasted.

If define statement is very long and we want to write in next line then end first line by \. For example:

```
#include<stdio.h>
#define word c is powerful language.
int main(){
printf("%s",word);
```

```
return 0;
}
```

Output: c is powerful language

1.1.15 NESTING OF MACROS:

We can also use one macro in the definition of another macro. That is macro definitions may be nested. Consider the following macro definitions

```
# define SQUARE(x)((x)*(x))
```

Undefining a macro:
A defined macro can be undefined using the statement

```
# undef identifier.
```

This is useful when we want to restrict the definition only to a particular part of the program.

1.1.16 FILE INCLUSION:

The preprocessor directive "#include file name" can be used to include any file in to your program if the function s or macro definitions are present in an external file they can be included in your file In the directive the filename is the name of the file containing the required definitions or functions alternatively the this directive can take the form

```
#include< filename >
```

Without double quotation marks. In this format the file will be searched in only standard directories. The c preprocessor also supports a more general form of test condition #if directive. This takes the following

```
#if constant expression
  {
     statement-1;
     statement-2;
       ....
       ....
   }
  #endif
```

the constant expression can be a logical expression such as test < = 3 etc

1.1.17 PREPROCESSOR OPERATORS IN C:

There are two operators in c preprocessor:
**1. # :** This operator is called **string zing operator** which convert any argument in the macro function in the string. So we can say pound sign # is string maker. Example

```
#include<stdio.h>
#define string(s) #s
int main(){
```

```
char str[15]= string(World is ours );
printf("%s",str);
return 0;
}
```

Output: World is ours

**Explanation** : Its intermediate file will look like:

```
int main(){
char str[15]=”World is our”;
printf("%s",str);
return 0;
}
```

Argument of string macro function ‘World is our’ is converted into string by the operator # .Now the string constant “World is our” is replaced the macro call function in line number 4.

**2. ## :** This operator is called **token pasting operator**. When we use a macro function with various argument then we can merge the argument with the help of ## operator. Example

```
#include<stdio.h>
#define merge(p,q,r) p##q##r
int main(){
int merge(a,b,c)=45;
printf("%d",abc);
return 0;
}
```

Output : 45

**Explanation :**

Arguments a,b,c in merge macro call function is merged in abc by ## operator .So in the intermediate file declaration statement is converted as :

```
int abc = 45;
```

1.1.18 CONDITIONAL DIRECTIVES IN C:

There are total six conditional compilation directives. There are:

- #if
- #elif
- #else
- #endif
- ifdef
- ifndef

#if directive :

It is conditional compilation directive. That is if condition is true then it will compile the c programming code otherwise it will not compile the c code.

Syntax 1:

```
#if <Constant_expression>
      ------------
      ------------
#endif
```

If constant expression will return 0 then condition will ture if it will return any non zero number condition will true.

Example:

```
#include<stdio.h>
#if 0
int main(){
     printf("HELLO WORLD");
return 0;
}
#endif
```

Output: Run time error, undefined symbol _main

Explanation: Due to zero as a constant expression in #if condition will false. So c code inside the #if condition will not execute. As we know without any main function we cannot execute any code.

We can also write #else in the #if directive. #else directive will only execute if condition is false.

**Syntax 2:**

```
#if <Constant_expression>
----------------
----------------
#else
----------------
----------------
#endif
```

Example:

```
#include<stdio.h>
#if(10)
int main(){

printf("errorandexception.blogspot.com");
return 0;
}
#else
int main(){
     We can write any thing
     5= a int;
     if(for(while(1);;));
          ++5;
```

```
                    10=24;
        return 0;
        }
        #endif
```

Output: errorandexception.blogspot.com.
Explanation: 10 is non zero integer constant. So #if condition is true.
Example:

```
#include<stdio.h>
#if -2
int main(){
printf("HELLO WORLD");
return 0;
}
#else
int main(){
printf("errorandexception.blogspot.com");
return 0;
}
#endif
```

Output: HELLO WORLD
Explanation: -2 is non zero number so #if condition is true.

Note: Consonant expression in #if condition should not include any c programming variable since all preprocessor directives execute just before the actual c code.
For example:

```
#include<stdio.h>
int main(){
int var = 5;
#if var

printf("errorandexception.blogspot.com");
#else
     printf("cquestionbank.blogspot.com");
#endif
return 0;
}
```

Output: cquestionbank.blogspot.com
Explanation: Directive #if will not think expression constant var as integer variable and also it will not throw an error. Then what is var for directive #if?

Directive #if will treat var as underfed macro constant. In c any underfed macro constant return zero so #else directive will execute. So proper way of above c code is:

```
#include<stdio.h>
#define var 10
int main(){
#if var

printf("errorandexception.blogspot.com");
#else
printf("cquestionbank.blogspot.com");
#endif
return 0;
}
```

Output: errorandexception.blogspot.com

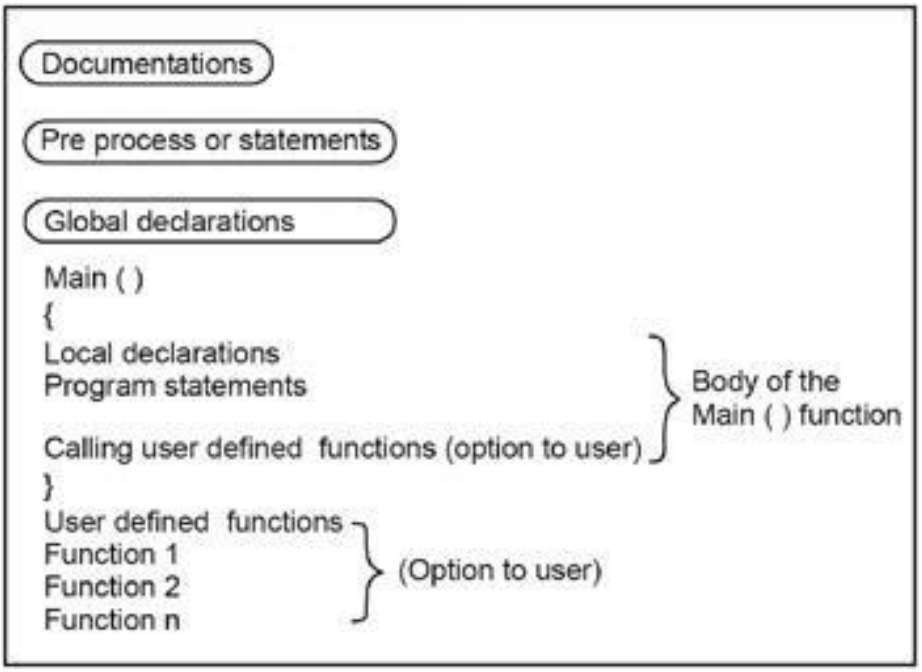
Explanation: Macro constant var will be replaced 10. Since 10 is non-zero number so #if part will execute.

Note: Constant expression in #if directive cannot be string constant. It can be character constant which returns its ASCII value to directive.

## 1.2 STRUCTURE OF A C PROGRAM AND DATA TYPES IN C

### 1.2.1 STRUCTURE OF C PROGRAM:

The structure of a C program is a protocol (rules) to the programmer, while writing a C program. The general basic structure of C program is shown in the figure below. The whole program is controlled within main ( ) along with left brace denoted by "{" and right braces denoted by "}". If you need to declare local variables and executable program structures are enclosed within "{" and "}" is called the body of the main function. The main ( ) function can be preceded by documentation, preprocessor statements and global declarations.




1.2.1 DOCUMENTATIONS:

The documentation section consist of a set of comment lines giving the name of the program, another name and other details, which the programmer would like to use later.

1.2.2 PREPROCESSOR STATEMENTS:

The preprocessor statements begin with # symbol and are also called the preprocessor directive. These statements instruct the compiler to include C preprocessors such as header files and symbolic constants before compiling the C program. Some of the preprocessor statements are listed below.

```
# include <stdio.h>
# include <math.h>
# include <stdlib.h>
# include <CONIO.h>
```

header files

```
# define P L 3.1412.
# define TRVE 1
# define FALSE Ø
```

Symbolic constants

1.2.3 GLOBAL DECLARATIONS:

The variables are declared before the main ( ) functions as well as user defined functions are called global variables. These global variables can be accessed by all the user defined functions including main ( ) function.

### 1.2.4 THE MAIN ( ) FUNCTION:

Each and Every C program should contain only one main ( ). The C program execution starts with main ( ) function. No C program is executed without the main function. The main ( ) function should be written in small (lowercase) letters and it should not be terminated by semicolon. Main ( ) executes user defined program statements, library functions and user defined functions and all these statements should be enclosed within left and right braces.

### 1.2.5 BRACES:

Every C program should have a pair of curly braces ({, }). The left braces indicates the beginning of the main ( ) function and the right braces indicates the end of the main ( ) function. These braces can also be used to indicate the user-defined functions beginning and ending. These two braces can also be used in compound statements.

### 1.2.6 LOCAL DECLARATIONS:

The variable declaration is a part of C program and all the variables are used in main ( ) function should be declared in the local declaration section is called local variables. Not only variables, we can also declare arrays, functions, pointers etc. These variables can also be initialized with basic data types.

For example,

```
main ( )
{
 int sum = 0;
 int x;
 float y;
}
```

Here, the variable sum is declared as integer variable and it is initialized to zero. Other variables declared as int and float and these variables inside any function are called local variables.

### 1.2.7 PROGRAM STATEMENTS:

These statements are building blocks of a program. They represent instructions to the computer to perform a specific task (operations). An instruction may contain an input-output statements, arithmetic statements, control statements, simple assignment statements and any other statements and it also includes comments that are enclosed within /* and */ . The comment statements are not compiled and executed and each executable statement should be terminated with semicolon.

### 1.2.8 USER DEFINED FUNCTIONS:

These are subprograms, generally, a subprogram is a function and these functions are written by the user are called user; defined functions. These functions are performed by user specific tasks and this also contains set of program statements. They may be written before or after a main () function and called within main () function. This is an optional to the programmer. Now, let us write a small program to display some message shown below.

```
#include <stdio.h>
main()
{
  printf ("welcome to the world of C/n");
}
```

### 1.2.9 DATATYPES IN C:

C language is rich in its data types. ANSI C supports four classes of data types:

- Primary data types (fundamental).
- User-defined data types.
- Derived data types.
- Empty data set (void).

#### 1.2.9.1 PRIMARY DATA TYPES:

All C compilers support four fundamental data types.

- Integer (int)
- Character (char)
- Floating point (float)
- Double-precision floating point (double).

Various data types and terminology used to describe is given below diagram.

**Integer type:**

| Signed type | Unsigned type |
|---|---|
| Int | Unsigned int |
| Short int | Unsigned short int |
| Long int | Unsigned long int |

**Character type:**

| |
|---|
| Signed char |
| Unsigned char |

**Floating Type**

| | | |
|---|---|---|
| Float | Double | Long double |

**Size and Range of Basic Data Types:**

| Data type | Range |
|---|---|
| Char | -128 to 127 |
| Int | -32,768 to 32,767 |
| Float | 3.4e-38 to 3.4e+38 |
| Double | 1.7e-308 to 1.7e+308 |

**Integer types:**

Integers are whole numbers with a range of values supported by a particular machine. Size of an integer that can be stored depends on the computer. If we use a 16-bit word length, the size of the integer value is limited to the range of ($-2^{n-1}$ to $+2^{n-1}-1$). A signed bit uses one bit for sign and 15 bits for the magnitude of the number. C has different forms of integer storage like **short int**, **long int**, in both **signed** and **unsigned** forms.

**Floating point types:**

Floating point numbers are stored in all machines, with 6 digits of precision. This data type is defined by using keyword ***float***. When the accuracy provided by float number is not sufficient, the type ***double*** can be used. The data type ***double*** can have precision of 14 digits. Further extension in precision we can use ***long double***.

**Character types:**

A single character can be defined as a character (***char***) type data. Characters are usually stored in 8 bits (one byte) of internal storage. While ***unsigned char*** have values between 0 to 255, ***signed char*** have values from -128 to 127.

1.2.9.2 USER-DEFINED DATA TYPES:

***User defined type declaration:***

In C language a user can define an identifier that represents an existing data type. The user defined datatype identifier can later be used to declare variables. The general syntax is typedef type identifier; here type represents existing data type and ‘identifier’ refers to the ‘row’ name given to the data type.

**Example:**

```
typedef  int  salary;
typedef  float  average;
```

Here salary symbolizes int and average symbolizes float. They can be later used to declare variables as follows:

```
Salary dept1, dept2;
Average section1, section2;
```

Therefore dept1 and dept2 are indirectly declared as integer datatype and section1 and section2 are indirectly float data type.

The second type of user defined data type is enumerated data type which is defined as follows.

```
enum identifier {value1, value2 …. Value n};
```

The identifier is a user defined enumerated datatype which can be used to declare variables that have one of the values enclosed within the braces. After the definition we can declare variables to be of this 'new' type as below.

```
enum identifier V1, V2, V3, …… Vn;
```

The enumerated variables V1, V2, ….. Vn can have only one of the values value1, value2 ….. value n

**The enum Data type:**

*enum* is the abbreviation for ENUMERATE, and we can use this keyword to declare and initialize a sequence of integer constants.

Here's an example:

```
enum colors {RED, YELLOW, GREEN, BLUE};
```

I've made the constant names uppercase, but you can name them whichever way you want. Here, *colors* is the name given to the set of constants - the name is optional. Now, if you don't assign a value to a constant, the default value for the first one in the list - *RED* in our case, has the value of *0*. The rest of the undefined constants have a value 1 more than the one before, so in our case, *YELLOW* is *1*, *GREEN* is *2* and *BLUE* is *3*.
But you can assign values if you wanted to:

```
enum colors {RED=1, YELLOW, GREEN=6, BLUE };
```

Now RED=1, YELLOW=2, GREEN=6 and BLUE=7.
The main advantage of *enum* is that if you don't initialize your constants, each one would have a unique value. The first would be zero and the rest would then count upwards.

```
#include <stdio.h>
int main() {
 enum  {RED=5,  YELLOW,  GREEN=4,
BLUE};

 printf("RED = %d\n", RED);
 printf("YELLOW = %d\n", YELLOW);
 printf("GREEN = %d\n", GREEN);
 printf("BLUE = %d\n", BLUE);
 return 0;
}
```

This will produce following results
RED = 5
YELLOW = 6
GREEN = 4
BLUE = 5

1.2.3 EMPTY DATA SET (VOID):

A function with void result type ends either by reaching the end of the function or by executing a return statement with no returned value. The void type may also appear as the sole argument of a function prototype to indicate that the function takes no arguments. Note that despite the name, in all of these situations, the void type serves as a unit type, not as a zero or bottom type, even though unlike a real unit type which is a singleton, the void type is said to comprise an empty set of values, and the language does not provide any way to declare an object or represent a value with type void.

## 1.3 MODIFIERS IN C AND TYPE CASTING

1.3.1 NEED OF MODIFIERS:

Let us consider an example of a program, which will accept an age from the user to do some processing. Because the age is represented in numbers, so we will have to use the integer data type int. We all know that even under exceptional case an age of a person cannot exceed more than 150. Now, that we are using an integer data type it occupies 2 Bytes of memory, which would not be required to represent the value 150. Instead the value 150 could easily be saved in an integer of 1 Byte in size, but the default size of an integer is 2 Bytes. So we have a problem here. Well, note that we can’t solve.

To override the default nature of a data type, C has provided us with data type modifiers as follows:

**Signed**:
By default all data types are declared as signed. Signed means that the data type is capable of storing negative values.

**Unsigned:**
To modify a data types behavior so that it can only store positive values, we require to use the data type unsigned. For example, if we were to declare a variable age, we know that an age cannot be represented by negative values and hence, we can modify the default behavior of the int data type as follows:

```
unsigned int age;
```

This declaration allows the age variable to store only positive values. An immediate effect is that the range changes from (-32768 to 32767) to (0 to 65536)

**Long:**
Many times in our programs we would want to store values beyond the storage capacity of the basic data types. In such cases we use the data type modifier long. This doubles the storage capacity of the data type being used.

E.g. long int annual salary will make the storage capacity of variable annual salary to 4 bytes. The exception to this is long double, which modifies the size of the double data type to 10 bytes. Please note that in some compilers this has no effect.

**Short:**
If long data type modifier doubles the size, short on the other hand reduces the size of the data type to half. Please refer to the example of age variable to explain the concept of data type modifiers. The same will be achieved by providing the declaration of age as follows:

```
short int age;
```

This declaration above, will provide the variable age with only 1 byte and its data range will be from -128 to 127.

### 1.3.2 TYPECASTING:

In computer science, **type conversion** or **typecasting** refers to different ways of, implicitly or explicitly, changing an entity of one data type into another. This is done to take advantage of certain features of type hierarchies or type representations. One example

would be small integers, which can be stored in a compact format and converted to a larger representation when used in arithmetic computations. C allows programmers to perform typecasting by placing the type name in parentheses and placing this in front of the value.

For instance

```
main()
{
float a;
a = (float) 5 / 3;
}
```

The above program gives result as 1.666666. This is because the integer 5 is converted to floating point value before division and the operation between float and integer results in float.

From the above it is clear that the usage of typecasting is to make a variable of one type, act like another type for one single operation. So by using this ability of typecasting it is possible for create ASCII characters by typecasting integer to its character equivalent.

Typecasting is also used in arithmetic operation to get correct result. This is very much needed in case of division when integer gets divided and the remainder is omitted. In order to get correct precision value, one can make use of typecast as shown in example above. Another use of the typecasting is shown in example below.

For instance:

```
main()
{
int a = 5000, b = 7000 ;
long int c = a * b ;
}
```

Here two integers are multiplied and the result is truncated and stored in variable c of type long int. But this would not fetch correct result for all. To get a more desired output the code is written as

```
long int c = (long int) a * b;
```

Though typecast has so many uses one must take care about its usage since using typecast in wrong places may cause loss of data like for instance truncating a float when typecasting to an int.

```
#include <stdio.h>
main()
   {
        double d1 = 1234.56;
        int i1=456;

        printf("The value of d1 as int without cast operator %d\n",d1);
        printf("The value of d1 as int with cast operator %d\n",(int)d1);
        printf("The value of i1 as double without cast operator %f\n",i1)
;
        printf("The value of i1 as double with cast operator %f\n",(doubl
e)i1);

        i1 = 10;
        printf("Effect of multiple unary operator %f\n",(double)++i1);
        i1 = 10;
        printf("Effect of multiple unary operator %f\n",(double)- ++i1);
        i1 = 10;
        printf("Effect of multiple unary operator %f\n",(double)- -i1);
        i1 = 10;
        printf("Effect of multiple unary operator %f\n",(double)-i1++);

   }
```

***Output:***

The value of d1 as int without cast operator 1889785610
The value of d1 as int with cast operator 1234
The value of i1 as double without cast operator 1234.559570
The value of i1 as double with cast operator 456.000000
Effect of multiple unary operator 11.000000
Effect of multiple unary operator -11.000000
Effect of multiple unary operator 10.000000
Effect of multiple unary operator -10.000000

1.3.3 IMPLICIT CONVERSION:

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type.
For example:

```
short  a=2000;
int b;
b=a;
```

Here, the value of ‘a’ has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler

can signal with a warning. This can be avoided with an explicit conversion.

### 1.3.4 EXPLICIT CONVERSION:

C is a strong-typed language. Many conversions, especially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

### 1.3.5 TYPE CONVERSIONS IN EXPRESSIONS:

C permits mixing of constants and variables of different types in an expression, but during evaluation it adhere to very strict rules of 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 the higher type. Given below is the sequence of rules that are applied while evaluating expressions.

- All short and char are automatically converted to int.
- If one of the operands is long double, the other will be converted to long double and the result will be long double.
- Else, if one of the operands is double, the other will be converted to double and the result will be double.
- Else, if one of the operands is float, the other will be converted to float and the result will be float.
- Else if one of the operand is unsigned long int, the other will be converted to unsigned long int and the result will be unsigned long int.

C automatically converts all floating point operands to double precision. The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment:

- Float to int causes truncation of the fractional part.
- Double to float causes rounding of digits.
- Long int to int causes dropping of the excess higher order bits.

## 1.4 VARIABLES, CONSTANTS AND ARRAYS

### 1.4 VARIABLES:

### 1.4.1 DEFINING VARIABLES

A variable is a meaningful name of data storage location in computer memory. When using a variable you refer to memory address of computer.

### 1.4.2 NAMING VARIABLES:

The name of variable can be called identifier or variable name in a friendly way. It has to follow these rules:

- The name can contain letters, digits and the underscore but the first letter has to be a letter or the underscore. Be avoided underscore as the first letter because it can be clashed with standard system variables.
- The length of name can be up to 247 characters long but 31 characters are usually adequate. Keywords cannot be used as a variable name.

Of course, the variable name should be meaningful to the programming context.

### 1.4.3 DECLARING VARIABLES:

To declare a variable you specify its name and kind of data type it can store. The variable declaration always ends with a semicolon.
For example:

```
int counter;
char ch;
```

You can declare variables at any point of your program before using it. The best practice suggests that you should declare your variables closest to their first point of use so the source code is easier to maintain. In C programming language, declaring a variable is also defining a variable.

### 1.4.4 INITIALIZING VARIABLES:

You can also initialize a variable when you declare it.
For example:

```
int x=10;
char ch='a';
```

As per the scope variables, the variables are divided into two types:

1. Global variable
2. Local variable

Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration. The scope of local variables is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function *main*) their scope is between its

declaration point and the end of that function. In the example above, this means that if another function existed in addition to *main*, the local variables declared in *main* could not be accessed from the other function and vice versa.

```
#include<stdio.h>
#include<iostream.h>
int Integer;
char acharacter;
char string[20];
unsigned int NumberOfSons//global variable declarations

int main()
{
unsigned short Age;
float Anumber, Anotherone;//local variable declaration
printf("enter your age");
scanf("%u",&age);
}
```

1.4.5 CONSTANTS:

A "constant" is a number, character, or character string that can be used as a value in a program. Use constants to represent floating-point, integer, enumeration, or character values that cannot be modified.

Constant Syntax:

*floating-point-constant*

*integer-constant*

*enumeration-constant*

*character-constant*

*The identifiers must conform to the following rules:*

1. First character must be an alphabet (or underscore)
2. Identifier names must consists of only letters, digits and underscore.
3. An identifier name should have less than 31 characters.
4. Any standard C language keyword cannot be used as a variable name.
5. An identifier should not contain a space.

The *const* keyword is to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

**Note:**

- You can declare the *const* before or after the type. Choose one and stick to it.
- It is usual to initialise a *const* with a value as it cannot get a value *any other way.*

The pre-processor **#define** is another more flexible method to define ***constants*** in a program.

```
#define TRUE            1
#define FALSE           0
#define NAME_SIZE       20
```

Here TRUE, FALSE and NAME_SIZE are constant

There are two different types of constants.

They are,

- Numeric constants
- Character constants

### 1.4.6 NUMERIC CONSTANTS:

In Numeric constants are again divided into two types.

They are,

1. Integer constants
2. Real Constants

**Integer constant:**

An Integer Constant refers to a sequence of digits. There are three types of integer, namely decimal, octal and hexadecimal. Decimal integers consists of a set of digital, 0 to 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are:

123

-321

0

654321

+78

Note: Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

15 750

20,000

$1000

are illegal numbers.

An octal integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integers are:

037

0

0435

0551

A sequence of digits preceded by 0x and Ox is considered as hexadecimal integers. They may also include alphabets A through F or a through F. The letters A through F represent the numbers 10 through 15.

0x2

0x9F

0xbcd

0x

Real constants:

Integer number is inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called real.

0.0083

-0.75

435.36

+247.0

1.4.7 CHARACTER CONSTANTS:

They are two types of character constants:

1. String constants
2. Backslash constants

**String constants:**

- A string constant may be a single character or multiple characters. A single character constant contains a single character enclosed within a pair of single quote marks.
  `'4','5'.....`

  `printf("%d",'a');`

  `output: 97`

  `printf("%c",97);`

  `output: a`

- A string constant is a sequence of character enclosed in double quotes. The character may be letters numbers, special characters and blank space.
  "Hello!"

  "1987"

  "WELL DONE"

  "?.....!"

  "5+3"

"X"

**Backslash Character constants:**

C supports some special backslash character constants that are used in output functions. The table will illustrates backslash characters:

| Constants | Meaning |
|---|---|
| '\a' | Audible alert(bell) |
| '\b' | Back space |
| '\f' | Form feed |
| '\n' | New line |
| '\r' | Carriage return |
| '\t' | Horizontal tab |
| '\v' | Vertical tab |
| '\'' | Single quote |
| '\"' | Double quote |
| '\?' | Question mark |
| '\\' | Backslash |
| '\0' | null |

1.4.8 ARRAYS :( SUBSCRIPTED VARIBLES):

Arrays are the group of related data items that share a common name and of them belong to the same data type. It is followed by [ ] (Index or subscripted elements of variables). An array in C Programming Language can be defined as number of memory locations, each of which can store the same data type and which can be references through the same variable name.

An array is a collective name given to a group of similar quantities. These similar quantities could be percentage marks of 100 students, number of chairs in home, or salaries of 300 employees or ages of 25 students. Thus an array is a collection of similar elements. These similar elements could be all integers or all floats or all characters etc. Usually, the array of characters is called a "string", where as an array of integers or float is called simply an array. All elements of any given array must be of the same type i.e. we can't have an array of 10 numbers, of which 5 are ints and 5 are floats.

Arrays and pointers have a special relationship as arrays use pointers to reference memory locations.

### 1.4.9 DECLARATION OF AN ARRAY:

Arrays must be declared before they can be used in the program. Standard array declaration is as

```
type variable_name [length of array];
```

Here type specifies the variable type of the element which is going to be stored in the array. In C programming language we can declare the array of any basic standard type which C language supports.
For example,

```
double height[10];
float width[20];
int min[9];
char name[20];
```

In C Language, an array starts at position 0. The elements of the array occupy adjacent locations in memory. C Language treats the name of the array as if it were a pointer to the first element This is important in understanding how to do arithmetic with arrays. Any item in the array can be accessed through its index, and it can be accessed anywhere from within the program. So,

```
m=height [0];
```

Variable m will have the value of first item of array height.

The program below will declare an array of five integers and print all the elements of the array.

```
int myArray [5] = {1,2,3,4,5};
/* To print all the elements of the array
for (int i=0;i<5;i++)
{
   printf("%d", myArray[i]);
}
```

### 1.4.10 INITIALIZING ARRAYS:

Initializing of array is very simple in c programming. The initializing values are enclosed within the curly braces in the declaration and placed following an equal sign after the array name. Here is an example which declares and initializes an array of five elements of type int. Array can also be initialized after declaration. Look at the following C code which demonstrates the declaration and initialization of an array.

```
int myArray[5] = {1, 2, 3, 4, 5};
int studentAge[4];
studentAge[0]=14;
studentAge[1]=13;
studentAge[2]=15;
studentAge[3]=16;
```

1.4.11 MULTI-DIMENSIONAL ARRAYS:

In C Language one can have arrays of any dimensions. To understand the concept of multidimensional arrays let us consider the following 4 X 5 matrix

| Row number (i) | Column numbers (j) | | | | | |
|---|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** | **4** |
| | **0** | 10 | 3 | 5 | -9 | -6 |
| | **1** | 5 | 6 | -8 | 7 | 24 |
| | **2** | -8 | 16 | 2 | 12 | 45 |
| | **3** | 10 | 13 | -10 | 4 | 5 |

Let us assume the name of matrix is X. To access a particular element from the array we have to use two subscripts on for row number and other for column number the notation is of the form X [i] [j] where i stands for row subscripts and j stands for column subscripts. Thus X [0] [0] refers to 10, X [2] [1] refers to 16 and so on In short multi dimensional arrays are defined more or less in the same manner as single dimensional arrays, except that for subscripts you require two squire two square brackets.

Below given are some typical two-dimensional array definitions

```
float table [50] [50];
char line [24] [40];
```

The first example defines tables as a floating point array having 50 rows and 50 columns. The number of elements will be 2500 (50 X 50). The second declaration example establishes an array line of type character with 24 rows and 40 columns. The number of elements will be (24 X 40).

Consider the following two dimensional array definition:

```
int values [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
Values [0] [0] = 1 Values [0] [1] = 2 Values [0] [2] = 3 Values [0] [3] =
4
Values [1] [0] = 5 Values [1] [1] = 6 Values [1] [2] = 7 Values [1] [3] =
```

```
8
Values [2] [0] = 9 Values [2] [1] = 10 Values [2] [2] = 11 Values [2] [3]
= 12
```

Here the first subscript stands for the row number and second one for column number. First subscript ranges from 0 to 2 and there are altogether 3 rows second one ranges from 0 to 3 and there are altogether 4 columns.

Alternatively the above definition can be defined and initialised as

```
int values [3] [4] = {
                        {
                                    1, 2, 3, 4
                        }
                        {
                                    5, 6, 7, 8
                        }
                        {
                                     9, 10, 11, 12
                        }
                       };
```

Here the values in first pair of braces are initialised to elements of first row, the values of second pair of inner braces are assigned to second row and so on. Note that outer pair of curly braces is required. If there are two few values within a pair of braces the remaining elements will be assigned as zeros.

## 1.5 OPERATORS IN C AND ITS PRECEDENCE

### 1.5.1 OPERATOR

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. C supports a rich set of operators. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical of logical expressions.

C operators can be classified into a number of categories. They include:

- Arithmetic
- Relational
- Logical
- Increment or Decrement

- Conditional
- Bitwise
- Assignment
- Special

### 1.5.2 ARITHMETIC OPERATORS:

C provides all the basic operators. The operators +, -, *, and / all work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by -1.

| Operator | Meaning |
|---|---|
| + | Addition Or Unary Plus |
| - | Subtraction Or Unary Minus |
| * | Multiplication |
| / | Division |
| % | Modulo Division |

Integer division truncates any fractional part. The modulo division produces the remainder of an integer division. Examples of arithmetic operators are:

```
a - b          a + b

a * b          a / b
```

Here **a** and **b** are called operands. The modulo division operator % cannot be used on floating point data. C does not have an operator for exponentiation.

**Integer Arithmetic:** When both the operands in a single arithmetic expression such as a + b are integers, the expression is called as integer expression, and the operation is called integer arithmetic. Integer arithmetic always yields an integer value. The largest integer depends on the machine. During integer division, if both the operands are of same sign, the result is

truncated towards zero, if one of them is negative; the direction of truncation is implementation dependent. That is,

```
6/7 = 0 and -6/-7 = 0
```

But -6/7 may be zero or -1(machine dependent)

Similarly during modulo division, the sign of the result is always the sign of the first operand (the dividend) this is

```
-14 % 3 = 2

-14 % -3 = -2

14 % -3 = 2
```

**Real Arithmetic:** An arithmetic operation involving only real operands is called real arithmetic. Any operand may have either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.

If x, y and z are floats, then we will have:

```
X = 6.0/7.0 = 0.857143

Y = 1.0/3.0 = 0.333333
```

The operator % cannot be used with real operands.

**Mixed_Mode Arithmetic:** When one of the operand is real and the other is integer, the expression is called mixed-mode arithmetic expression. If either operand is of the real type, then only the real expression is performed and the result is always a real number. Thus

```
15/10 = 1.5
```

Where as

```
15/10 = 1
```

If the numerator term is less then the denominator term (b<a), then the following rules are applied.

1. b % a gives b
2. (-b) % a gives (-b)
3. (-b) % (-a) gives (-b)
4. b%(-a) gives (-a)

If the division has both the numerator and denominator are real values then the result is also real. If it is mixed that means one is 'int' and second is in 'float' then also the result is 'real'.

The “+,-,*,and %” are divided into two types.

They are,

1. low priority group and
2. high priority group

The low priority group is “+” and “-”.

The high priority group is “*” and “/”.

### 1.5.3 RELATIONAL OPERATORS:

We often compare two quantities, and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of relational operators. Any expression which contains relational operators is called relational expressions. The value of relational expression is either one or zero. If the specified relation is true then result is one. If the specified relation is false then result is zero. C supports six relational operators in all.

| operator | meaning |
|---|---|
| < | is less than |
| <= | is less than or equal to |
| > | is greater than |
| >= | is greater than or equal to |
| == | is equal to |
| != | is not equal to |

A simple relational expression contains only one relational operator takes the following form:

```
Arithmetic exp-1   relational operator   Arithmetic exp-2
```

Some examples of simple relational expressions and their values:

5.8 >= 2 TRUE

7 <= 9 TRUE

-90 >= 1 FALSE

3 != 4 TRUE

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators. Relational operators are used in decision statements such as, if and while to decide the course of action of a running program.

### 1.5.4 LOGICAL OPERATORS:

An expression which combines two or more relational expressions is termed as a logical expression or a compound relational expression. A logical expression yields a value of one or zero.

| Operator | Meaning |
|---|---|
| && | Logical AND |
| \|\| | Logical OR |
| ! | Logical NOT |

Truth table:

| operand-1 | operand-2 | operand-1 && operand-2 | operand-1 \|\| operand-2 |
|---|---|---|---|
| non-zero | non-zero | 1 | 1 |
| non-zero | 0 | 0 | 1 |
| 0 | non-zero | 0 | 1 |
| 0 | 0 | 0 | 0 |

### 1.5.5 ASSIGNMENT OPERATORS:

Assignment operators are used to assign the result of an expression of a variable. In C has a set of ‘shorthand’ assignment operators of the form

```
v op = exp;
```

Where v is the variable, exp is an expression and op is a C binary arithmetic operator. The operator '=' is known as the shorthand assignment operator.

the assignment statement

```
v op = exp;
```

Is equivalent to

```
v = v op(exp);
```

Consider an example x += y+1;

This is same as the statement x = x+(y+1). The shorthand operator += means 'add y+1 to x' or increment x by 'y+1'.

Table: Shorthand assignment operators

| statement with simple assignment operator | statement with shorthand operator |
|---|---|
| a = a + 1 | a += 1 |
| a = a -1 | a -= 1 |
| a = a * (n +1) | a *= n+1 |
| a = a/ (n+1) | a /= n+1 |
| a = a %b | a %=b |

The Use of Shorthand Assignment Operators has three advantages:

- What appears on the left-hand side need not be replaced and therefore if becomes easier to write
- The statement is more concise and easier to read.
- The statement is more efficient.

Example program:

```
main()
{
      int i= 0, n, sum = 0;
      printf("enter a value");
      scanf("%d", &n);
```

OUTPUT: enter a value 3

Sum value is 3

1.5.6 INCREMENT OR DECREMENT OPERATOR:

C has two very useful operators not generally found in other languages. These are the increment and decrement operators:

++ and --

The operator ++ adds 1 to the operand while -- subtracts 1. Both are unary operators takes the following form:

++m; or m++;

--m; or m--;

++m; is equivalent to m = m+1;(or m+=1;)

--m; is equivalent to m = m-1; (or m-=1;)

We use increment and decrement statements in for and while loops extensively.

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
```

In this case, the value of t and m would be 6. Suppose, if we write the above statements as

```
m = 5;
```

Then the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left then increments the operand.

Example program:

```
main()
{
      int i=10, n, m, x, y;
      n = i++;
      printf("i value is %d", i);
      m = ++i;
      printf("i value is %d", i);
      x = i--;
      printf("i value is %d", i);
      y = --i;
      printf("i value is %d", i);
}
```

OUTPU

i value is 11

i value is 10

1.5.7 CONDITIONAL OPERATOR:

A ternary operator ' ? :" is available in C to construct conditional expressions of the form

```
exp1 ? exp 2 : exp 3
```

Where exp1, exp2, and exp3 are expressions. The operator? : works as follows: exp1 is evaluated first. If it is non-zero (true), then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either exp2 or exp3) is evaluated.

Example program:

```
main()
{
      int a, b, c;
      printf("enter two values");
```

OUTPUT: enter two values 10 20

bigger value is 20.

### 1.5.8 BIT WISE OPERATOR:

C has a distinction of supporting special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double.

| Operator | Meaning |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise exclusive OR |
| << | Shift left |
| >> | Shift right |
| ~ | One's compliment |

Table: Bitwise operators

BITWISE AND: Bitwise AND is denoted by the symbol '&' (ampersand). The & operator performs a bitwise AND on two integers. Each bit in the result is 1 only if both corresponding bits in the two input operands are 1. The following is a chart that defines &, defining AND on individual bits.

| x | y | x & y |
|---|---|---|

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

BITWISE OR: Bitwise OR is denoted by the symbol '|' (vertical bar). The | (vertical bar) operator performs a bitwise OR on two integers. Each bit in the result is 1 if either of the corresponding bits in the two input operands is 1. The following chart that defines |, defining OR on individual bits.

| x | y | x \| y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

BITWISE XOR: Bitwise XOR is denoted by the symbol '^' (caret). The ^ (caret) operator performs a bitwise exclusive-OR on two integers. Each bit in the result is 1 if one, but not both, of the corresponding bits in the two input operands is 1. The following chart that defines Bitwise XOR

| x | y | x ^ y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

SHIFT LEFT: The << operator shifts its first operand left by a number of bits given by its second operand, filling in new 0 bits at the right.

SHIFT RIGHT: The >> operator shifts its first operand right. If the first operand is unsigned, >> fills in 0 bits from the left, but if the first operand is signed, >> might fill in 1 bits if the high-order bit was already 1 .

ONE's COMPLIMENT: The ~ (tilde) operator performs a bitwise complement on its single integer operand. Complementing a number means to change all the 0 bits to 1 and all the 1s to 0s.

1.5.9 SPECIAL OPERATORS:

C supports some special operators of interest such as comma operator, size of operator, pointer operators (& and *) and member selection operators (. And ->).

THE COMMA OPERATOR: The comma operator can be used to link the related expressions together. A comma-linked list of expressions is evaluated left to right and the value of right-most expression ids the value of the combined expression. For example the statement

```
value = (x=10, y=5, x+y);
```

First assigns the value 10 to x, then assigns 5 to y, and finally assigns 15(10+ 5) to value. Since comma operator has lower precedence of all operators, the parentheses are necessary. Some applications of comma operator:

In for loops:

```
for(n = 1, m = 10; n <= m; n++, m++)
```

in while loops:

```
while(c = getchar(), c!='10')
```

exchanging values:

```
t = x, x = y, y = t;
```

THE SIZEOF OPERATOR: The size of is a complier time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

```
m = sizeof(sum);

n = sizeof(long int)
```

The size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

### 1.5.10 OPERATOR PRECEDENCE AND ASSOCIATIVITY:

Each operator in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of the levels. The operators at the higher level of precedence are evaluated first. The operators are of same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as the associativity property of an operator. The groups are listed in the order of decreasing precedence (rank 1 indicates the highest precedence level and 15 the lowest)

### 1.5.11 OPERATOR PRECEDENCE TABLE:

| Operator | Description | Associativly | Prority |
|---|---|---|---|
| ( )<br>[ ] | Function call<br>Subscript | Left to Right<br>Left to Right | 1<br>1 |
| !<br>~<br>+<br>-<br>++<br>--<br>*<br>&<br>(Type)<br>size of | Logical Negation<br>Bit wise Not<br>Uniary plus<br>Uniary minus<br>Increment<br>Decrement<br>Directional<br>Address<br>type casting<br>size of an object | Right to Left<br>“<br>“<br>“<br>“<br>“<br>“<br>“<br>“<br>“ | 2<br>2<br>2<br>2<br>2<br>2<br>2<br>2<br>2<br>2 |
| *<br>/<br>% | Multiplication<br>Division<br>Modulo Division | Left to Right<br>“<br>“ | 3<br>3<br>3 |
| +<br>- | Addition<br>Subtraction | Left to Right<br>“ | 4<br>4 |

| | | | |
|---|---|---|---|
| <<<br>>> | Left shift<br>Right shift | Left to Right<br>“ | 5<br>5 |
| < or <=<br>> or >= | Less than<br>Greater than | Left to Right<br>“ | 6<br>6 |
| <br>!= | Equity<br>Not equal | Left to Right<br>“ | 7<br>7 |
| & | Bit wise And | Left to Right | 8 |
| | Bit wise Or | Left to Right | 9 |
| ^ | Bit wise exclusive Or | Left to Right | 10 |
| && | Logical And | Left to Right | 11 |
| | Logical Or | Left to Right | 12 |
| ?: | Conditional | Left to Right | 13 |
| Op | Assignment<br>Operational Assignment (+=,-=,*=,/=,%=) | Right To Left<br>“ | 14<br>14 |
| , | Coma | Left to right | 15 |

Consider the following statement:

If (x ==10 + 15 && y<10)

The precedence rule say that the addition operator has the higher priority and the logical operator (&&) and the relational operator (++ and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to:

If (x == 25 && y < 10)

The next step is to determine whether x is equal to 25 and y is less than 10. If we assume a value of 20 for x and 5 for y, then

x == 25 is false (0)

y < 10 is true (1)

Note that since the operator < enjoys a higher priority compared to +++, y, 10 is tested first and then x == 25 is tested. Finally we get:

if( FALSE && TRUE)

Because one of the conditions is FALSE, the complex condition is FALSE.

### 1.5.12 PRECEDENCE OF ARITHMETIC OPERATORS:

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

| | |
|---|---|
| High priority | * ? % |
| Low priority | + - |

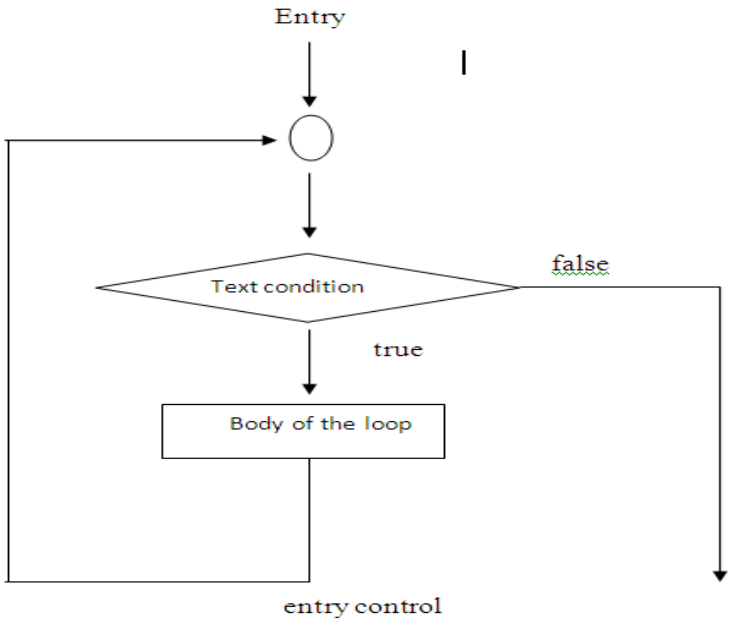
The basic evaluation procedure includes two left to right passes through the expression. During the first pass, the high priority operator (if any) is applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered.

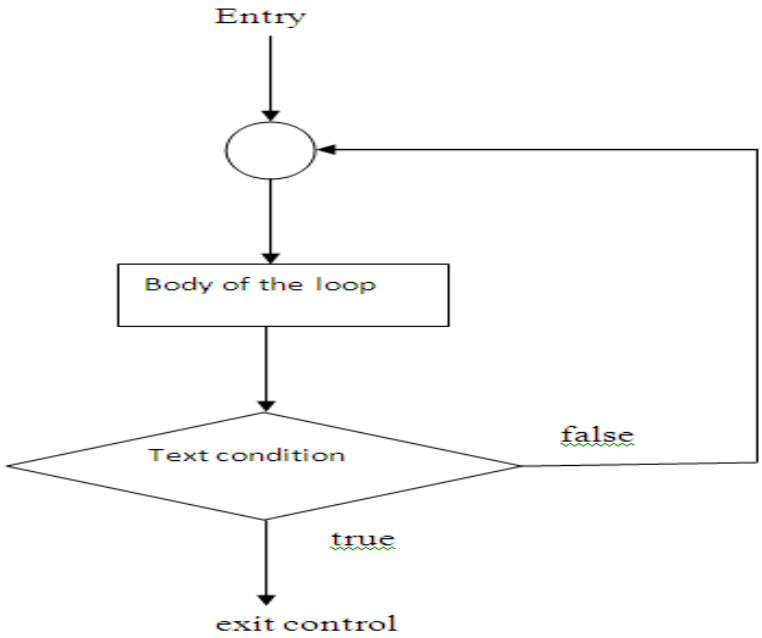
## 1.6 LOOPING STRUCTURES IN C

### 1.6.1 INRODUCTION:

In looping, a sequence of statements is executed until some conditions for termination of the loop are satisfied. A program loop therefore consists of two segments, one known as the body of the loop and the other known as the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop. Depending on the position of the control statement in the loop, a control structure may be classified either as the entry-controlled loop or as the exit controlled loop.

In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.




The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test conditions will eventually transfer the control out

of the loop, due to some reason it does not do so, the control sets up an infinite loop and the body is executed over and over again.
A looping process, in general, would include the following four steps:

- Setting and initialization of a counter.
- Execution of the statements in the loop.
- Test for a specified conditions for execution of the loop.
- Incrementing the counter.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.
The C language provides for three loop constructs for performing loop operations. They are:

- The **while** statement.
- The **do** statement.
- The **for** statement.

### 1.6.2 THE WHILE STATEMENT:

The simplest of all the looping structures in C is the while statement. We have used while in many of programs. The basic format of while statement is:

```
While (test
condition)

{

Body of the loop

}
```

The **while** is an entry-cont xt condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test-condition is once again is evaluated and if it is true, the body is executed once again. This process is continues until the test condition becomes false and the control is transferred out of the loop. On exit, the program continues with the statements immediately after the body of the loop.

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

The body of the loop is executed 10 times for n = 1,2,3….10 each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as n<11; the result would be the same.
Another example of while statement which uses the keyboard input is shown below:

```
char = ' ';
while(character != 'y')
char = getchar();
```

First the character ...ment then begins by testing whether character is not equal to Y. since the character was initialized to ' ', the test is true and the loop statement char = getchar(); is executed. Each time a letter is keyed in, the test is carried out and the loop statement is executed until the letter Y is pressed. When Y is pressed, the condition becomes false and the loop terminates.
Example program:

```
main()
{
int n;
printf("enter a value");
scanf("%d", &n);
i = n;
while(i>=1)
     {
       printf("%d", i);
        i--;
     }
}
```

OUTPUT: E...
6

1.6.3 THE DO STATEMENT:

The while loop construct that makes a test of condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the conditions is not satisfied at the very first attempt. But in some situations it is necessary to execute the body of the loop before the condition is tested. This can be done by using **do** statement.

```
do
{
    body of the loop
}while(test-condition);
```

On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test-condition is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues until the condition becomes false and control will be transferred to statement after to the while. **do..while** is anexit- controlled loop and therefore the body of the loop is always executed at least once.

Example program:

```
main()
{
      int n,i;
      printf("enter a value");
      scanf("%d", &n);
      i=1;
do
{
       printf("%d",i);
       i++;
}while(i<=n);
}
```

OUTPUT: E
1

The test conditions may have compound statements also. For instance, the statement

```
while (number >0 && number < 100)
```

1.6.4 THE FOR STATEMENT:

The **for** loop is another entry-controlled loop that provides a best looping control structure. The general form of the **for** loop is:

```
for (initialization ; test-condition ; increment)
{
      body of the loop;
}
```

The execution of the **for** statement is as follows:

- Initialization of the control variables is done first, using assignment statements such as **i=1** and **count = o**. the variables I and count are known as loop-control variables.
- The value of the control variable is tested using **test-condition**. The test-condition is a relational expression, such as i<10 that determines when the loop is exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and control is transferred to out of the loop.
- When the body of the loop is executed, the control is transferred to the **for** statement for evaluating the last statement in the loop. The incremented statement is i=i+1 (assignment statement) executed and the new value is tested in the condition statement of the **for** loop. If the condition fails the control is transferred to next statement.

Consider the following program:

```
for(i=0; i<=9; i++)
    {
        printf("%d", i);
    }
    printf("\n");
```

This for loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. There is no semicolon at the end of the increment section, i = i+1

The **for** statement allows for negative statements.

```
for(i= 9; i>=0; i = i-1)
{
    printf ("%d",i);
}
printf("\n");
```

This loop is exec 0 instead of 0 to 9. Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example:

```
for(i= 9; i<9; i = i-1)

    printf ("%d",i);
```

Will never be executed because the test condition fails at the very beginning itself. One of the important points about the foe loop is that all the three actions namely initialization, testing and increments, are placed in the for loop statement itself, thus making them visible to the programmers, and users in one place

Table: comparison of the three loops

| For | while | do |
|---|---|---|
| for(n=1;n<=10;n++)<br>{<br>……..<br>} | n = 1;<br>while(n<=10)<br>{<br>…….<br>n= n+1;<br>} | n =1;<br>do<br>{<br>……<br>n= n+1;<br>}while(n<=10); |

1.6.5 ADDITIONAL FEATURES OF for LOOP:

The **for** loop in C has several capabilities that are not found in other constructs. For example, more than one variable can be initialized at a time in the **for** statement.

The statements:

```
p =1;
for (n=0; n<17; ++n)
```

Can be rewritten as: note that initialization section has two parts p = 1 and n =1 separated by a comma.

```
for (p=1, n=0; n<17; ++n)
```

We can have also more than one increment section in one part. For example, the loop is perfectly valid. The multiple arguments in the increment section are separated by commas.

```
for(n=1, m=2; n<=m; n=n+1, m=m-1)
{
     printf("%d %d %d\n", n, m);
}
```

The third feature ne testing need not be limited only to the loop control variable. Consider the example:

```
sum =o;
for (i=1; i<20 && sum<100; ++i)
{
sum = sum + i;
printf ("%d %d\n", sum);
}
```

The loop uses a compound test condition with a control variable **i** and external variable **sum**. The loop is executed as long as both the conditions i<20 and sum < 100 are true. The **sum** is evaluated inside the loop. It is also permissible to use expression in the assignment statements of initialization and increment sections. For example, a statement of type is perfectly valid.

```
for( x = (m+n)/2; x>0; x = x/2)
```

Another unique aspect of **for** loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```
.....
.....
m = 5;
for( ; m != 100; )
{
      printf("%d\n", m);
      m = m + 5;
}
```

Both initialization and [illegible] statement. The initialization has been done before the for statement and the control variable is incremented inside the loop. In the above case the increment and initialization are left blank. However, the semicolons separating the sections must remain. If the test condition is not present, the for statement sets up and infinite loop. Such loops can be broken using **break** or **goto** statements in the loop.

We can set up **time delays** using the **null statement** as follows:

```
for(j =1000; j > 0; j = j-1);
```

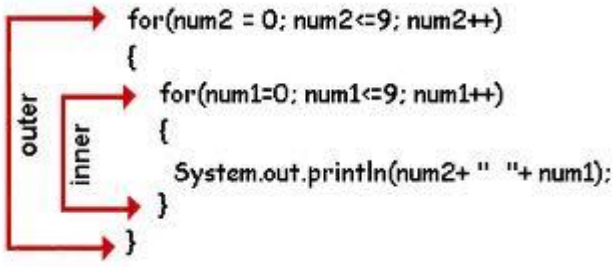
This loop is exe[illegible]ime delay. Notice that there is no body of the loop called null statement. Above loop can also be written as

```
for(j =1000; j > 0; j = j-1);
```

This implies that the C complier will not give an error message if we place a semicolon by mistake at the end of a **for** statement. The semicolon will be considered as null statement and the program may produce some nonsense.

### 1.6.6 NESTING OF for LOOPS:

Nesting of loops, that is, one for statement within another for statement, is allowed in C. for example:




Example program:

```
main()
{
      int i, n, x, j;
      printf("enter a number");
      scanf("%d", &n);
      for(i=1; x=1; i<=n; i++)
      {
            for(j=1; j<=i; j++)
            printf("%d", x++);
            printf("\n");
      }
}
```

Output:

```
Enter the value 4
1
2 3
4 5 6
7 8 9 10
```

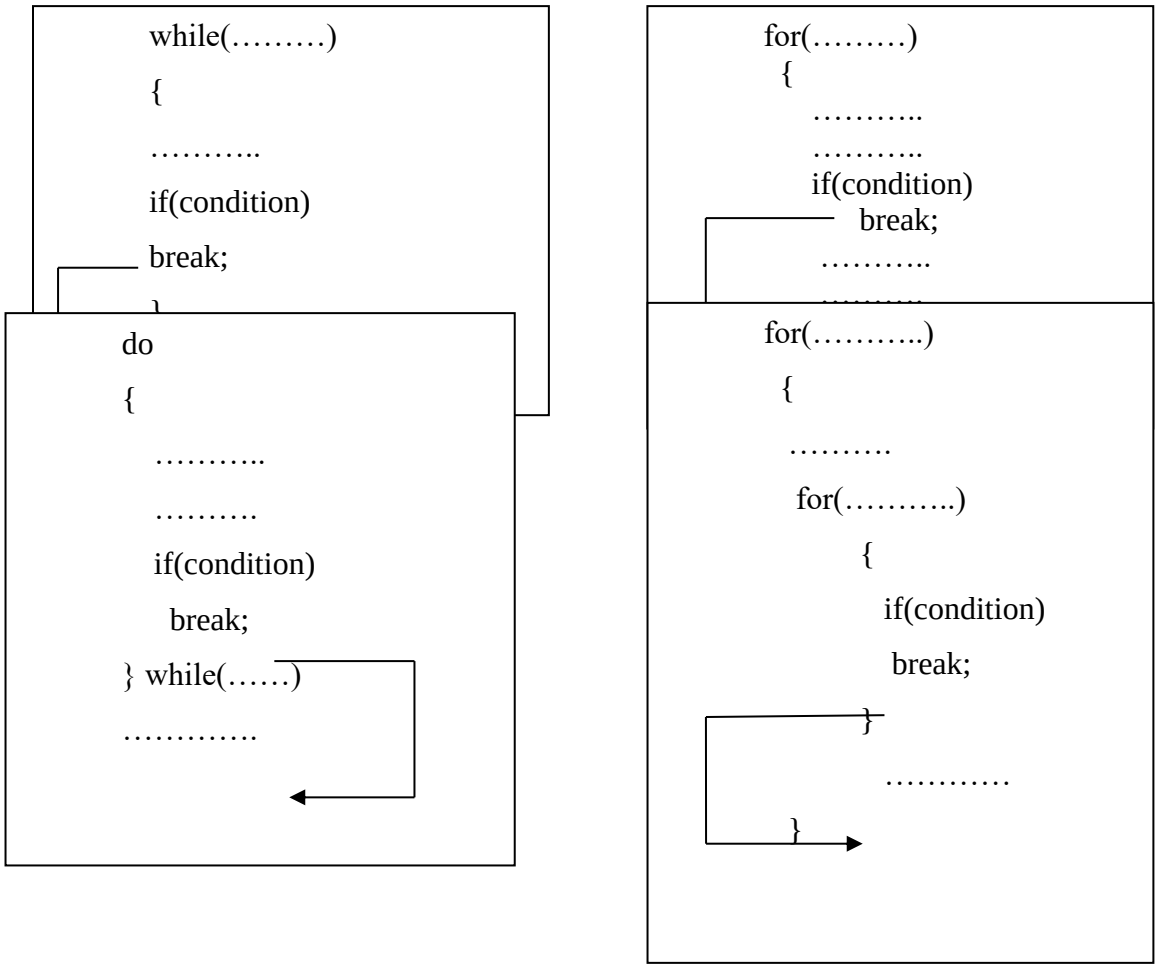
1.6.7 JUMPS IN L

Loops perfo ntrol variable fails to satisfy the test- condition. ' ided in advance and the test condition is written he loop it becomes necessary to skip a part of the ondition occurs. C permits a jump from one state mp out of loop. For example, consider a case of s ram loop written for reading and testing the names a 100 times must be terminated as soon as the desired name is found.

1.6.8 JUMPING OUT OF A LOOP:

An early exit from a loop can be accomplished by using the **break** statement or the **goto** statement. We use break and goto statements in both control structures and looping

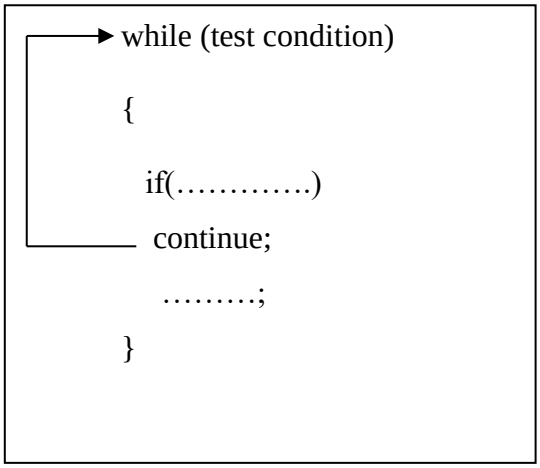
structures. When the **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the **break** statement would only exit from the loop containing it. That is, **break** is exit only a single loop.

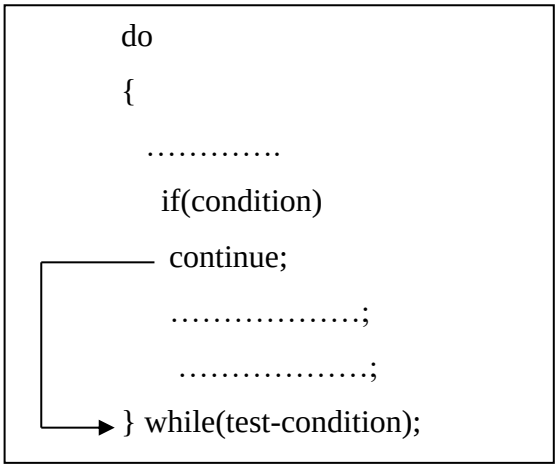


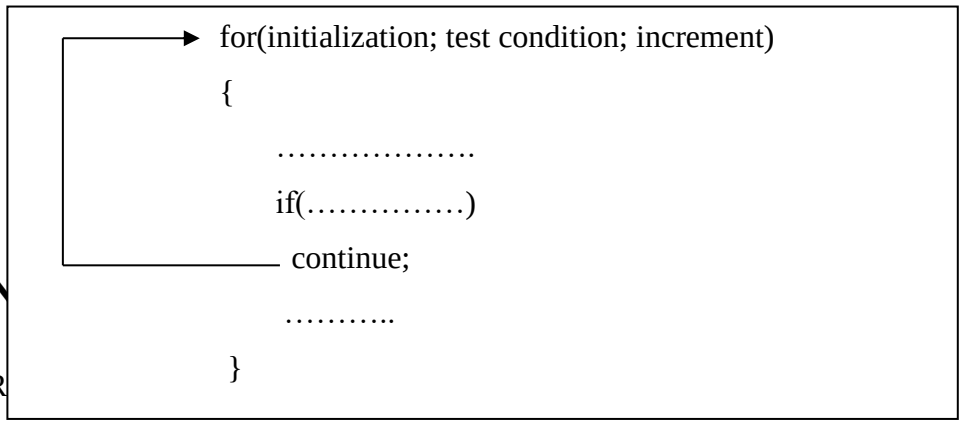

1.6.9 SKIPPING A PART OF A LOOP:

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category. On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the break statement, C supports another similar statement called the **continue** statement. However, unlike the break which causes the loop to be terminated, the **continue**, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the complier, "SKIP THE FOLLOWING STATEMENT AND CONTINUE WITH NEXT OTERATION". In **while** and **do** loops, **continue** causes the control to go directly to the test condition and then to continue the iteration process. In the case of **for** loop, the increment section of the loop is executed before the test-condition is evaluated.




## 1.7 FUN

### 1.7.1 INTR

A function is a block of code that has a name and it has a property that it is reusable i.e. it can be executed from as many different points in a C Program as required. Function groups a number of program statements into a unit and gives it a name. This unit can be invoked from other parts of a program. A computer program cannot handle all the tasks by itself. Instead its requests other program like entities – called functions in C – to get its tasks done. A function is a self contained block of statements that perform a coherent task of same kind

The name of the function is unique in a C Program and is Global. It names that a function can be accessed from any location within a C Program. We pass information to the

function called **arguments** specified when the function is called. And the function either returns some value to the point it was called from or returns nothing. We can divide a long C program into small blocks which can perform a certain task. A function is a self contained block of statements that perform a coherent task of same kind.

1.7.2 STRUCTURE OF A FUNCTION:

All functions have the form:

```
function name(argument list)
argument declaration;
{
      local variable declarations;
      executable statement1;
      executable statements2;
```

All parts are e argument list and its associated argument declaration parts are optional. The return statement is the mechanism for returning a value to the calling function. This is also an optional statement. Its absence indicates that function does not return any value.

**Function Header:**

In the first line of the above code

int sum(int x, int y)

It has three main parts

1. The name of the function i.e. *sum*
2. The parameters of the function enclosed in parenthesis
3. Return value type i.e. *int*

**Argument List:** The argument list contains valid variable names separated by commas. The list must be surrounded by parentheses. Note that no semicolon follows the closing parenthesis. The argument variables receive values from the calling function, thus providing a means for data communications from the calling function to the called function. Some examples of functions with arguments are:

```
quadratic(a,b,c)
mul(a,b)
```

All arguments variables must be declared for their types after the function header and before the opening brace of the function body.

**Function Body:** Whatever is written with in { } in the above example is the body of the function.

**Function Prototypes:** The prototype of a function provides the basic information about a function which tells the compiler that the function is used correctly or not. It contains the same information as the function header contains. The prototype of the function in the above example would be like

```
int sum (int x, int y);
```

The only difference between the header and the prototype is the semicolon; there must the semicolon at the end of the prototype. Functions groups a number of program statements into a unit and gives it a name. This unit can be invoked from other parts of a program. A computer program cannot handle all the tasks by itself. Instead its requests other program like entities – called functions in C – to get its tasks done. A function is a self contained block of statements that perform a coherent task of same kind. e.g.

**Simple Functions:** Our first example demonstrates a simple function those purpose is to print a line of 45 asterisks. The example program generates a table, and lines of asterisks are used to make the table more readable. Here's the listing for Table:

**Why we use functions?**

The most important reason to use functions is to aid in the conceptual organization of a program. Another reason to use functions is to reduce program size. Any sequence of instructions that appears in program more than once is a candidate for being made into a function. The function's code is stored in only one place in memory, even though the function is executed many times in the course of the program.

**Two reasons**

1. Writing functions avoids rewriting the same code over and over.
2. Using functions it becomes easier to write programs and keep track of what they are doing.

**Advantages:**
It is easy to use.

- Debugging is more suitable for programs.
- It reduces the size of a program.
- It is easy to understand the actual logic of a program.
- Highly suited in case of large programs.
- By using functions in a program, it is possible to construct modular and structured programs.

**Function Declaration:** Just as you can't use a variable without first telling the compiler what it is, you also can't use functions without telling the compiler about it, There are two ways to do this. The approach we show here its o declare the function before it is called. The other approach is to define it before it's called. ; we'll examine that next.) in the Table program, the functions star line() is declared in the line.

Void star line ( );

The declaration tells the compiler that at some later point we plan to present a function called star line. The keyword void specifies that the function has no return value, and the empty parentheses indicate that it takes no arguments.

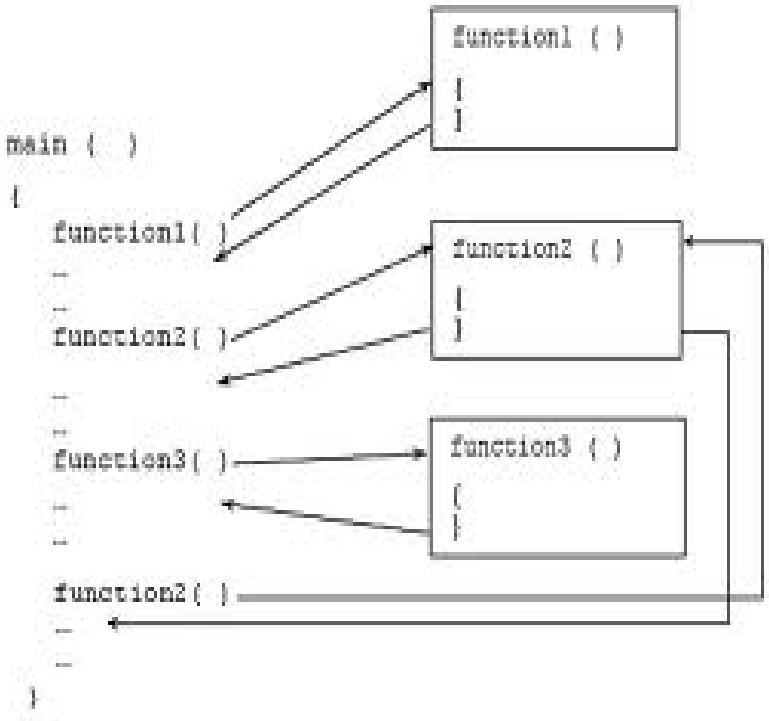
Notice that the functions declarations is terminated with a semicolon It is a complete statement in itself. Function declarations are also called prototypes, since they provide a model or blueprint for the function. They tell the compiler," a function that looks like this is coming up later in the program, so it's all right if you see references to it before you see the function itself."

**Calling the Function:** The function is called (or invoked) three times from main (). Each of the three calls look like this:

Star line():

This is all we need to call a function name, followed by parentheses. The syntax of the call is very similar to that of declaration, except that the return type is not used. A semicolon terminates the call. Executing the call statement causes the function to execute; that is, control is transferred to the function, the statement in the function definition are executed, and then control returns to the statement following the function call.

Ex:




Note: main is also a function which is used to indication to complier to where from the completion should start.

The function is basically divided into two types. They are

1. User defined and
2. Pre-defined functions

The main distinction between user-defined and library function is that the former are not required to be written by user while the latter have to be developed by the user at the time of writing a program

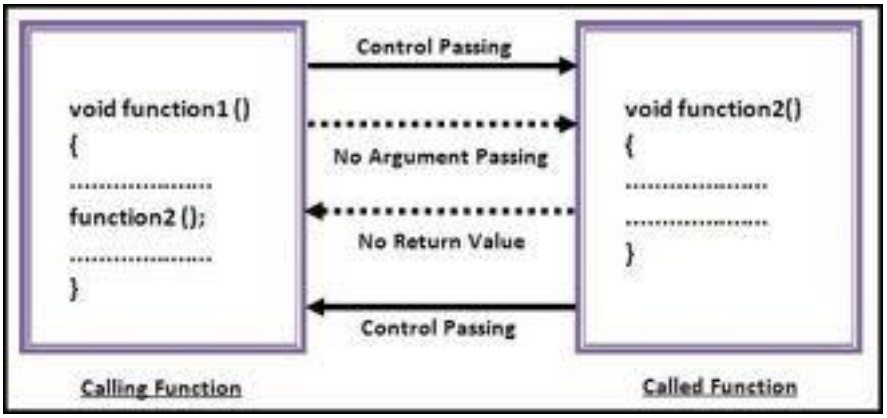
**User Defined Functions:** A User-Defined Function, or UDF, is a function provided by the user of a program or environment, in a context where the usual assumption is that functions are built into the program or environment. The data communication from calling function to called function can be achieved by parameters or arguments. The arguments and parameters are equal but the variables present at the calling function are called Actual parameters and variables present at the function definition then they are called formal parameters. The actual parameters may be constants, variables (or) expressions. The formal parameters must be variables. We can pass only one value to the definition of the function to called function by using return values.

Based on the arguments and parameters they are divided into three types. They are

1. Function with no arguments and no return values,

2. Function with arguments and no return values and
3. Function with arguments and return values.

**Function with no arguments and no return values:** In this we not pass any arguments to definition of the function and we do return any value to the calling function.

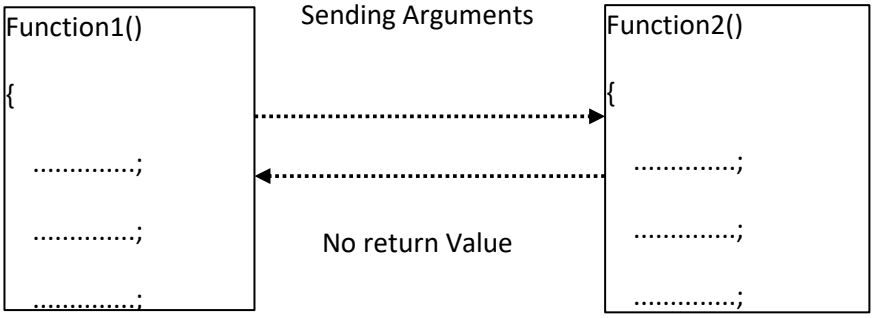



Ex:

```
main()
      {
            sum();
      }
      void sum()
      {
            int a,b;
            printf("Enter any 2 no's:");
            scanf("%d",&a,&b);
            printf("sum of numbers:%d", a+b);

      }
```

The above function not sending any arguments to function definition and not receiving anything from function definition.

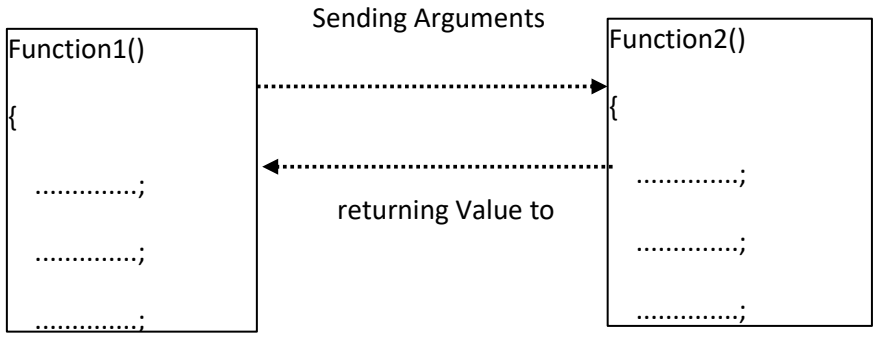
**Function with arguments and no return values:** In this we are passing arguments to defination of the function and we are not going to return any value to the calling function.
Eg:



```
{
int a,b;
printf("Enter any 2 no's:");
scanf("%d",&a,&b);
sum(a,b);      // sending arguments to function definition

}
void sum(int a, int b)   // a and b local variables to function
definition
{
     printf("sum of numbers:%d",a+b);
}
```

The above function we are sending arguments as a & b to function definition and are received by formal parameters a and b and not returning to any information to the called function.

**Function with arguments and return values:** In this we are passing arguments to definition of the function and we are going to return the result of function definition to the calling function.




Ex:

```
main()
{
     int a,b,c;
     printf("Enter any 2 no's:");
     scanf("%d",&a,&b);
     c=sum(a,b);    // sending arguments to function
                         definition
     printf("Sum of two numbers:%d",c);
}
void sum(int a, int b)// a and b local variables to
                         function definition
{
     return(a+b);   // sending result to the called function
```

```
}
```

The above function we are sending arguments as a & b to function definition and are received by formal parameters a and b and returning the result to called function and received by the variable C at to the called function.

### 1.7.3 HANDLING OF NON-INTEGER FUNCTIONS:

C function returns a value of the type as the default case when no other type is specified explicitly. For example,

return (sum);

Returns only the integer part of sum. This is due to the absence of the type-specifier in the function header. In this case, we can accept the integer value of sum because the truncated decimal part is insignificant compared to the integer part. We must do two things to enable a calling function to receive a non-integer value from a called function:

- The explicit type specifier, corresponding to the data type required must be mentioned in the function header. The general form of the function definition is:

```
Type-specifier function-name (argument list)

argument declaration;

{

      function statements;

}
```

- The type spe
- The called function must be declared at the start of the body in the calling function, like any other variable. This is to tell the calling function the type of data that the function is actually returning.

Example program:

```
main()
{
      float a, b, mul();
      double div();
       a = 12.345;
       b = 9.82;
      printf("%f\n", mul(a,b));
      printf("%f\n", div(a,b));
}
float mul(x,y)
float x,y;
```

```
      {
            return(x * y);
      }
      double div(p,q)
      double p,q;
      {
            return(p/q);
}
```

The declaration part of main function declares not only variables but the functions mul and div as well. This only tells the complier that mul will return a float type value and div a double type value Parentheses that follow mul and div specify that they are functions instead of variables. If we mismatch between the type of data that the called function returns and the type of data that the calling function expects, we will have unpredictable results.

1.7.5 FUNCTIONS RETURNING NOTHING:

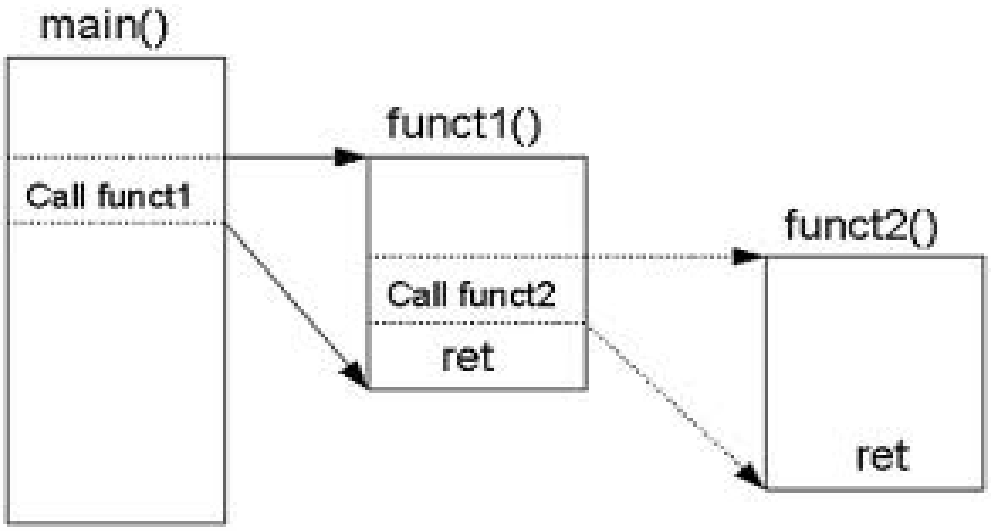
Consider the program:

```
      main()
      {
            printline();
            value();
            printline();
      }
      printline();
      {
            ..........
      }
      value()
      {
            ..........
      }
```

T refore they were not decl re them in them main wi turn values. This prevents

### 1.7.6 NESTING OF FUNCTIONS:

C permits nesting of functions, main can call function1, which calls function2, which calls function3 and so on. Consider the following example:




```
main()
{
  int a,b,c;
  float ratio();
  scanf("%d%d%d",&a,&b,&c);
  printf("%fn",ratio(a,b,c));
}
float ratio(x,y,z)
int x,y,z;
{
  if(difference(y,z))
  return(x/y-z));
  else
  return(0,0);
}
difference(p,q)
{
  int p,q;
    {
     if(p!=q)
     return(1);
     else
     return(0);
    }
```

The above program calculates the ratio a/(b-c)

And prints the result. we have the following three functions:

main ()
ratio ()
difference ()

Main reads the values of a, b and c and calls the function ratio to calculate the value a/(b-c). This ratio cannot be evaluated if (b-c) = o. Therefore, ratio calls another function difference to test whether the difference (b-c) is zero or not. Difference returns 1, if b is not equal to c; otherwise returns zero to the function ratio. In turn, ratio calculates the value a/(b-c) if it receives 1 returns the result in float. In case, ratio receives zero form difference, it sends back 0.0 to main including that (b-c) = 0.

### 1.7.7 RECURSIVE FUNCTION:

When a called function in turn calls another function a process of ‘chaining’ occurs. When a function calls itself then it is called as 'recursive function’.

```
main()
{
        printf(“this an example of recursion \n”);
        main();
}
```

When execut

This is an example of recursion
This is an example of recursion
This is an ex

Execution is terminated abruptly; otherwise the execution will continue indefinitely.
Another example of recursion is the evaluation of factorials of a given number. The factorial of a number n is expressed as a series of repetitive multiplications.

**Features:**

- There should be at least one if statement used to terminate recursion.
- It does not contain any looping statements.

**Advantages:**

- It is easy to use.
- It represents compact programming structures.

**Disadvantages:**

- It is slower than that of looping statements because each time function is called.

1.7.8 FUNCTIONS WITH ARRAYS:

In functions also we can pass the values of an array. To pass an array to a called function, it is sufficient to list the name of the array, without any subscripts, and the size of an array as arguments. For example, the call largest (a, n)

Will pass all the elements contained in the array a of size n. The called function excepting this call must be appropriately defined. The largest function header might look like;

```
float largest (array, size)
float array[];
int size;
```

The function largest is defined to take two arguments. The array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

```
float array[];
```

The pair of brackets informs the complier that the argument array is an array of numbers. It is not necessary to specify the size of the array here.

Example program:

```
main()
{
      float largest();
      static float value[4] = {2.5, -4.75,
1.2, 3.67};
      printf("%f\n", largest(value,4));
}
float largest(a,n)
float a[];
int n;
{
      int i;
      float max;
      max = a[0];
      for(i=1; i<n; i++     )
            if(max < a[i])
            max = a[i];
      return(max);
}
```

When th ments of the array value are pass on. The largest function finds the largest value in the array and returns the result to the main.

1.7.9 SPECIAL LIBRARY FUNCTIONS:

C provides a repertoire of standard library functions and macros for specialized purposes (and for the advanced user). These may be divided into various categories. For instance

- Character identification (ctype.h)
- String manipulation (string.h)
- Mathematical functions (math.h)

A program generally has to #include special header files in order to use special functions in libraries. The names of the appropriate files can be found in particular compiler manuals. In the examples above the names of the header files are given in parentheses.

1.7.10 CHARACTER IDENTIFICATION:

Some or all of the following functions/macros will be available for identifying and classifying single characters. The programmer ought to beware that it would be natural for many of these facilities to exist as macros rather than functions, so the usual remarks about macro parameters apply. An example of their use is given above. Assume that `true' has any non-zero, integer value and that `false' has the integer value zero. ch stands for some character, or char type variable.

```
isalpha(ch)
```

This returns true if ch is alphabetic and false otherwise. Alphabetic means a..z or A..Z.

```
isupper(ch)
```

Returns true if the character was upper case. If ch was not an alphabetic character, this returns false.

```
islower(ch)
```

Returns true if the character was lower case. If ch was not an alphabetic character, this returns false.

```
isdigit(ch)
```

Returns true if the character was a digit in the range 0..9.

isxdigit(ch)

Returns true if the character was a valid hexadecimal digit: that is, a number from 0..9 or a letter a..f or A..F.

```
isspace(ch)
```

Returns true if the character was a white space character, that is: a space, a TAB character or a newline.

```
ispunct(ch)
```

Returns true if ch is a punctuation character.

```
isalnum(ch)
```

Returns true if a character is alphanumeric: that is, alphabetic or digit.

```
isprint(ch)
```

Returns true if the character is printable: that is, the character is not a control character.

```
isgraph(ch)
```

Returns true if the character is graphic. i.e. if the character is printable (excluding the space)

```
iscntrl(ch)
```

Returns true if the character is a control character. i.e. ASCII values 0 to 31 and 127.

```
isascii(ch)
```

Returns true if the character is a valid ASCII character: that is, it has a code in the range 0..127.

```
iscsym(ch)
```

Returns true if the character was a character which could be used in a C identifier.

```
toupper(ch)
```

This converts the character ch into its upper case counterpart. This does not affect characters which are already upper case, or characters which do not have a particular case, such as digits.

```
tolower(ch)
```

This converts a character into its lower case counterpart. It does not affect characters which are already lower case.

```
toascii(ch)
```

This strips off bit 7 of a character so that it is in the range 0..127: that is, a valid ASCII character.

```
        main
           {
        char ch;
        printf ("VALID CHARACTERS FROM isalpha()\n\n");
        for (ALLCHARS)
               {
              if (isalpha(ch))
                                 {
                        printf ("%c ",ch);
         }
         }
printf ("\n\nVALID CHARACTERS FROM isupper()\n\n");
        for (ALLCHARS)
             {
              if (isupper(ch))
           {
              printf ("%c ",ch);
        }
             }
printf ("\n\nVALID CHARACTERS FROM islower()\n\n");
        for (ALLCHARS)
           {
              if (islower(ch))
            {
               printf ("%c ",ch);
            }
             }
printf ("\n\nVALID CHARACTERS FROM isdigit()\n\n");
        for (ALLCHARS)
           {
              if (isdigit(ch))
            {
```

```
                    printf ("%c ",ch);
                  }
                 }
       printf ("\n\nVALID CHARACTERS FROM isxdigit()\n\n");
              for (ALLCHARS)
                 {
                    if (isxdigit(ch))
                 {
                   printf ("%c ",ch);
                 }
                        }
               printf      ("\n\nVALID      CHARACTERS      FROM
ispunct()\n\n");
              for (ALLCHARS)
                {
                    if (ispunct(ch))
                {
                   printf ("%c ",ch);
                }
                }
       printf ("\n\nVALID CHARACTERS FROM isalnum()\n\n");
              for (ALLCHARS)
               {
                    if (isalnum(ch))
                {
                  printf ("%c ",ch);
                                    }
                    }
       printf ("\n\nVALID CHARACTERS FROM iscsym()\n\n");
              for (ALLCHARS)
              {
                    if (iscsym(ch))
                {
                                      printf ("%c ",ch);
                                   }
                     }
}
```

**Program Output:**

VALID CHARACTERS FROM isalpha()

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

VALID CHARACTERS FROM isupper()

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

VALID CHARACTERS FROM islower()

a b c d e f g h i j k l m n o p q r s t u v w x y z

VALID CHARACTERS FROM isdigit()

0 1 2 3 4 5 6 7 8 9

VALID CHARACTERS FROM isxdigit()

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

VALID CHARACTERS FROM ispunct()

! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~

VALID CHARACTERS FROM isalnum()

0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

VALID CHARACTERS FROM iscsym()

0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z

1.7.11 STRING MANIPULATION:

The following functions perform useful functions for string handling.

**strcat()**

This function "concatenates" two strings: that is, it joins them together into one string. The effect of: To join two static strings together, the following code is required:

```
char *s1 = "string one";
char *s2 = "string two";
main ()
 {
  char buffer[255];
  strcat(buffer,s1);
   strcat(buffer,s2);
 }
```

**strlen()**

This function returns a type int value, which gives the length or number of characters in a string, not including the NULL byte end marker. An example is:

```
int len;
char *string;
 len = strlen (string);
```

**strcpy()**

This function copies a string from one place to another. Use this function in preference to custom routines: it is set up to handle any peculiarities in the way data are stored. An example is

```
char *to,*from;
 to = strcpy (to,from);
```

Where to is a pointer to the place to which the string is to be copied and from is the place where the string is to be copied from.

**strcmp()**

This function compares two strings and returns a value which indicates how they compared. An example:

```
int value;
char *s1,*s2;
value = strcmp(s1,s2);
```

The value returned is 0 if the two strings were identical. If the strings were not the same, this function indicates the (ASCII) alphabetical order of the two. s1 > s2, alphabetically, then the value is > 0. If s1 < s2 then the value is < 0. Note that numbers come before letters in the ASCII code sequence and also that upper case comes before lower case. There are also

variations on the theme of the functions above which begin with strn instead of str. These enable the programmer to perform the same actions with the first n characters of a string:

**strncat()**

This function *concatenates* two strings by copying the first n characters of this to the end of the onto string.

```
char *onto,*new,*this;
new = strncat(onto,this,n);
```

**strncpy()**

This function copies the first n characters of a string from one place to another

```
char *to,*from;
int n;
to = strncpy (to,from,n);
```

**strncmp()**

This function compares the first n characters of two strings

```
int value;
char *s1,*s2;
value = strcmp(s1,s2,n);
```

The following functions perform conversions between strings and floating point/integer types, without needing to use scanf(). They take a pre-initialized string and work out the value represented by that string.

**atof()**

ASCII to floating point conversion.

```
double x;
char *stringptr;
x = atof(stringptr);
```

**atoi()**

ASCII to integer conversion.

```
int i;
char *stringptr;
 i = atoi(stringptr);
```

**atol()**

ASCII to long integer conversion.

```
long i;
char *stringptr;
i = atol(stringptr);
```

1.7.13 MATHEMATICAL FUNCTIONS:

C has a library of standard mathematical functions which can be accessed by #including the appropriate header files (math.h etc.). It should be noted that all of these functions work with double or long float type variables. All of C's mathematical capabilities are written for long variable types. Here is a list of the functions which can be expected in the standard library file. The variables used are all to be declared long

```
int i;              /* long int */
double  x, y, result;      /* long float */
```

The functions themselves must be declared long float or double (which might be done automatically in the mathematics library file, or in a separate file) and any constants must be written in floating point form: for instance, write 7.0 instead of just 7.

**ABS()**

MACRO. Returns the unsigned value of the value in parentheses. See fabs() for a function version.

**fabs()**

Find the absolute or unsigned value of the value in parentheses:
result = fabs(x);

**ceil()**

Find out what the ceiling integer is: that is, the integer which is just above the value in parentheses. This is like rounding up.

```
i = ceil(x);   /* ceil (2.2) is 3 */
```

**floor()**

Find out what the floor integer is: that is, the integer which is just below the floating point value in parentheses

```
i = floor(x);  /* floor(2.2) is 2 */
```

**exp()**

Find the exponential value.

```
result = exp(x);
result = exp(2.7);
```

**log()**

Find the natural (Naperian) logarithm. The value used in the parentheses must be unsigned: that is, it must be greater than zero. It does not have to be declared specifically as unsigned. e.g.

```
result = log(x);
result = log(2.71828);
```

**log10()**

Find the base 10 logarithm. The value used in the parentheses must be unsigned: that is, it must be greater than zero. It does not have to be declared specifically as unsigned.

```
result = log10(x);
result = log10(10000);
```

**pow()**

Raise a number to the power.

```
result = pow(x,y); /*raise x to the power y */
result = pow(x,2); /*find x-squared */
result = pow(2.0,3.2); /* find 2 to the power 3.2 ...*/
```

**sqrt()**

Find the square root of a number.

```
result = sqrt(x);
result = sqrt(2.0);
```

**sin()**

Find the sine of the angle in radians.

```
result = sin(x);
result = sin(3.14);
```

**cos()**

Find the cosine of the angle in radians.

result = cos(x);
result = cos(3.14);

**tan()**

Find the tangent of the angle in radians.

result = tan(x);
result = tan(3.14);

**asin()**

Find the arcsine or inverse sine of the value which must lie between +1.0 and -1.0.

result = asin(x);
result = asin(1.0);

**acos()**

Find the arccosine or inverse cosine of the value which must lie between +1.0 and -1.0.

result = acos(x);
result = acos(1.0);

**atan()**

Find the arctangent or inverse tangent of the value.

result = atan(x);
result = atan(200.0);

**atan2()**

This is a special inverse tangent function for calculating the inverse tangent of x divided by y. This function is set up to find this result more accurately than atan().

result = atan2(x,y);
result = atan2(x/3.14);

**sinh()**

Find the hyperbolic sine of the value. (Pronounced "shine" or "sinch")

result = sinh(x);
result = sinh(5.0);

**cosh()**

Find the hyperbolic cosine of the value.

result = cosh(x);
result = cosh(5.0);

**`tanh()`**

Find the hyperbolic tangent of the value.

```
result = tanh(x);
result = tanh(5.0);
```

1.7.14 MATHS ERRORS:

Mathematical functions can be delicate animals. There exist mathematical functions which simply cannot produce sensible answers in all possible cases. Mathematical functions are not "user friendly"! One example of an unfriendly function is the inverse sine function asin(x) which only works for values of x in the range +1.0 to -1.0. The reason for this is a mathematical one: namely that the sine function (of which asin() is the opposite) only has values in this range. The statement y **= asin (25.3);**
is nonsense and it cannot possibly produce a value for y, because none exists. Similarly, there is no simple number which is the square root of a negative value, so an expression such as: **x = sqrt(-2.0);** would also be nonsense. This doesn't stop the programmer from writing these statements though and it doesn't stop a faulty program from straying out of bounds. What happens then when an erroneous statement is executed? Some sort of error condition would certainly have to result.

In many languages, errors, like the ones above, are terminal: they cause a program to stop without any option to recover the damage. In C, as the reader might have come to expect, this is not the case. It is possible (in principle) to recover from any error, whilst still maintaining firm control of a program.

Errors like the ones above are called domain errors (the set of values which a function can accept is called the domain of the function). There are other errors which can occur too. For example, division by zero is illegal, because dividing by zero is "mathematical nonsense" - it can be done, but the answer can be all the numbers which exist at the same time! Obviously a program cannot work with any idea as vague as this. Finally, in addition to these "pathological" cases, mathematical operations can fail just because the numbers they deal with get too large for the computer to handle, or too small, as the case may be.

*Domain error*
Illegal value put into function
*Division by zero*
Dividing by zero is nonsense.
*Overflow*
Number became too large
*Underflow*

Number became too small.
*Loss of accuracy*
No meaningful answer could be calculated

Errors are investigated by calling a function called matherr(). The mathematical functions, listed above, call this function automatically when an error is detected. The function responds by returning a value which gives information about the error. The exact details will depend upon a given compiler. For instance a hypothetical example: if the error could be recovered from, matherr() returns 0, otherwise it returns -1. matherr() uses a "struct" type variable called an "exception" to diagnose faults in mathematical functions. This can be examined by programs which trap their errors dutifully. Information about this structure must be found in a given compiler manual.

Although it is not possible to generalize, the following remarks about the behavior of mathematical functions may help to avoid any surprises about their behavior in error conditions.

- A function which fails to produce a sensible answer, for any of the reasons above, might simply return zero or it might return the maximum value of the computer. Be careful to check this. (Division by zero and underflow probably return zero, whereas overflow returns the maximum value which the computer can handle.)
- Some functions return the value NaN. Not a form of Indian unleavened bread, this stands for `Not a Number', i.e. no sensible result could be calculated.
- Some method of signaling errors must clearly be used. This is the exception structure (a special kind of C variable) which gives information about the last error which occurred. Find out what it is and trap errors!
- Obviously, wherever possible, the programmer should try to stop errors from occurring in the first place.

# 1.8: STORAGE CLASSES IN C

1.8.1 INTRODUCTION:

Variables in C differ in behavior from those in most other languages. For example, in a BASIC program, a variable retains its value throughout the program. It is not always the case in C. it all depends on the ‘storage’ class a variable may assume.
A variable in C can have any one of the four storage classes:

- Automatic variables
- External variables
- Static variables

- Register variables

The scope of variable determines over what part(s) of the program a variable is actually available for use (active). Longevity refers to the period which a variable retains a given value during execution of a program (alive). So longevity has a direct effect on the utility of a given variable. The variables may also be broadly categorized, depending on the place of their declaration, as ***internal*** (local) or ***external*** (global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

1.8.2 AUTOMATIC VARIABLES:

Automatic variables are declared inside a function in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited. Hence the name automatic. Automatic variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred as ***local*** or ***internal*** variables. A variable declared inside a function without storage class specification is, by default, an automatic variable.

For instance, the storage class of the variable number in the below example:

```
main()
      {
      int number;
      ----------;
      }
```

We may also use the keyword **auto** to declare automatic variables explicitly.

```
main ()
      {
      auto int number;
      --------------------;
      }
```

One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other functions in the program. This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the complier. Illustrate example:

```
main ()
      {
            int m=1000;
            function2 ();
```

```
        printf("%d\n", m);
}
        function1 ()
        {
        int m=10;
        printf ("%d \n", m);
        }
        function2 ()
        {
        int m=100;
        function1 ();
        printf ("%d\n", m);
        }
```

**m** is an automatic variable and it is declared at the beginning of each function. **m** is initialized to 10, 100 and 1000 in **function1**, **function2** and **main** respectively. When executed, **main** calls **function2** which in turn calls **function1**. When **main** is active, m = 1000; but when **function2** is called, the **main**'s m is temporarily put on the shelf and the new local m =100 becomes active. Similarly, when **function1** is called, both the pervious values of m are put on the shelf and the latest value if **m** (=10) becomes active. As soon as function1 (m = 10) is finished, **function2** (m = 100) takes over again. As soon as it is done, **main** (m = 1000) takes over. The output clearly shows that the value assigned to m in one function does not affect its value in other functions and the local value of m is destroyed when it leaves a function.

OUTPUT:

10
100
1000

There are two consequences of the scope and longevity of **auto** variables. First, any variable local to main will normally live throughout the whole program, although it is active only in main. Secondly, during recursion, the nested variables are unique auto variables. Automatic variables can also be defined within a set of braces known as "blocks". They are meaningful only inside the blocks where they are defined. Automatic variables are meaningful only inside blocks where they are defined

```
main ()
     {
     int n, a, b;
     ……..;
     ……..;
     if (n<=100)
          {
```

```
            int n, sum;
            ……
            }
            ……/* sum is not valid here*/
      }
```

The variables **n, a** and **b** defined in **main** have scope from the beginning to the end of **main**. But the variable **n** defined in the **main** cannot enter into the block of scope level 2 because the scope level 2 contains another variable named **n**. the second **n** (which takes precedence over first **n**) is available only inside the scope level 2 and no longer available the moment control leaves the **if** block. If no variable **n** is defined inside the **if** block, the **n** defined in the **main** would be available inside the scope level 2 as well. The variable **sum** defined in the **if** block is not available outside the block.

1.8.3 EXTERNAL VARIABLES:

Variables that are both active and alive throughout the entire program are known as external variables, they are also known as global variables. Unlike local variables, global variables can be accessed by any function. External variables are declared outside the function. For example:

```
int number;
float length = 7.5;
main ()
      {
      ……..
      …….
      }
function1 ()
      {
      …….
      …….
      }
function2 ()
      {
      …….
      …….
      }
```

The variables number and length are available for use in all three functions. In case a local variable and a global variable have same name, the local variable will have precedence

over the global variable one in the function where it is declared. Consider the following example:

```
int count;
main()
        {
        count = 10;
        ……..
        ……
        }
function()
        {
        int count = 0;
        ……..
        count = count +1;
        }
```

When the **function** references the variable **count**, it will be referencing only its local variable, not the global one. The value of **count** in the **main** will not be affected. Once a variable has been declared as global, any function can use it and change its value. Then, subsequent functions can reference only that new value. Because of this property, we should try to use global variables only for tables or for variables shared between functions when it is convenient to pass them as parameters. Other aspect of a global variable is that it is visible only to the point of declaration to the end of the program. Consider a program segment:

```
main()
      {
      y=5;
      …..
      …..
      }
      int y;
      func1 ()
            {
            y = y+1;
            }
```

As far as **main** is concerned, y is not defined. So, the complier will issue an error message. Unlike local variables, global variables are initialized to zero by default. The statement
y = y+1; in **func1 ()** will assign 1 to y.

EXTERNAL DECLARATION:

In the program segment above the main cannot access the variable y as it has been declared after the main program. This problem can be solved by declaring the variable with the storage class **extern.** For example:

```
main()
     {
     extern int y; /* external declaration*/
     ….
     ….
     }
func1()
     {
     extern int y; /* external declaration*/
     …….
     }
int y; /*definition */
```

Although the variable y has been defined after both functions, the external declaration of y inside the functions informs the complier that y is an integer type defined somewhere else in the program. Note that the extern declaration does not allocate storage space for variables. In case of arrays, the definition should include their size as well.

MULTIFILE PROGRAMS:

So far we have been assuming that all the functions (including main) are defined in one file. However in real life programming environment we may use more than one source files which may be complied separately and linked later to form an executable object code. This approach is very useful because any change in one file does not affect other files thus eliminating the need for recompilation of the entire program.

Multiple source files can share a variable provided it is declared as an external variable appropriately. Variables that are shared by two or more files are global variables and therefore we must declare them accordingly in one file and then explicitly define them with extern in other files.

file1.c

```
main()
{
   extern int m;
```

file2.c

```
int m /*global variable*/
function2 ()
{
```

```
   int i;                                    int i;
   …….                                       ……..
   ……..                                      ……..
}                                          }
function1 ()                               function3 ()
{                                          {
   int j;                                     int count;
}                                          }
```

Use of **EXTERN** in multi files programs

The function **main** in **file1** can reference the variable **m** that is declared as global in file2. Remember, **function1** cannot access the variable m. if, however, the extern int m; statement is placed before **main**, and then both the functions could refer to variable to **m**. this can also be achieved by using **extern int m**; statement inside each function in **file1**.

The **extern** specify tells the complier that the following variable data types and names have already been declared elsewhere and no need to create storage space for them. It is responsibility to the linker to resolve the reference problem. It is important to note that a multi file global variable should be declared without **extern** in one of the files. The **extern** declaration is done in places where secondary references are made. If we declare as global in two different files used by a single program, then the linker will have a conflict as to which variable to use and, therefore, issues a warning.

1.8.3 STATIC VARIABLES:

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared static using the keyword **static** like. A static variable may be either an internal type or an external type, depending on the place of declaration. Internal static variables are those which are declared inside a function. The scope of internal static variable extends up to the end of the function. Therefore, internal static variable are similar to auto variables, except that they remain in existence (alive) throughout the remainder of the program. Therefore internal static variables can be used to retain values between function calls. A static variable is initialized once, when the program is complied. It is never initialized again.

Illustration of static variable:

```
main ()
     {
     int i;
     for (i=1;i<=3;i++)
```

```
stat ();
}
stat ()
{
static int x=0;
x=x+1;
printf ("x = %d\n", x);
}
```

OUTPUT:
x = 1
x = 2
x = 3

During the first call to **stat, x** is incremented to 1. Because **x** is static, this value persists and therefore, the next call adds another 1 to **x** giving it a value of 2. The value of x becomes three when the third call is made. If we declared x as auto variable then the output will be:

x = 1
x = 1
x = 1

This is because each time **stat ()** is called, the **auto** variable x is initialized to zero. When the function terminates, its value of 1 is lost.

An **external static** variable is declared outside of all functions and is variable to all the functions in that program. The difference between a external **static** variable and the simple static variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.
It is also possible to control the scope of a function. For example, we would like a particular function accessible only to the functions in the file in which it is defined, and not to any function in other files. This can be accomplished by defining 'that' function with the storage class **static.**

1.8.4 REGISTER VARIABLES:

We can tell the complier that a variable should be kept in one of the machine's registers, instead of keeping in the memory (where normal variables are stored). Since a register access is much faster than a memory access, keeping the frequently accessed variables in the register will lead to faster execution of programs.

**register int** count;

Although, ANSI standard does not restrict its application to any particular data type, most compliers allow only int or char variables to be placed in the register. Since only few variables can be placed in the register, it is important to carefully select the variable for this purpose. However, C will automatically convert **register** variables into non register variables once the limit is reached.

Table: Scope and Lifetime of Declarations

| storage class | declaration | visibility(active) | lifetime(active) |
|---|---|---|---|
| None | before all functions in a file | entire file plus other files where variables is declared with **extern.** | entire program(global) |
| **Extern** | before all functions in a file ( cannot be initialized) | entire file plus other files where variable is declared **extern** and the file where originally declared as global | global |
| **Static** | before all functions in a file | only in that file | global |
| none or **auto** | inside a function(only a block) | only in that function or block | until end of function or block |
| **register** | inside a function | only in that function or block | until end of function or block |

# 1.9: POINTERS

1.9.1 INTRODUCTION:

Pointers are another important feature of C language**.** Although they may appear a little confusing for a beginner, they are a powerful tool and handy to use once they are mastered. There are number of reasons for using pointers.

- A pointer enables us to access a variable that is defined outside the function.
- Pointers are more efficient in handling data tables.
- Pointers reduce length and complexity of a program.
- They increase the execution speed.
- The use of a pointer array to character strings results in saving of data storage space in memory.

1.9.2 UNDERSTANDING POINTERS:

A pointer is a variable which holds the address of the storage location for another given variable. C provides two operators & and * which allow pointers to be used in many versatile ways.

**& and ***

The & and * operators have already been used once to hand back values to variable parameters. They can be read in a program to have the following meanings:

& the address of...
* The contents of the address held in...

This reinforces the idea that pointers reach out an imaginary hand and point to some location in the memory and it is more usual to speak of pointers in this way. The two operators * and & are always written in front of a variable, clinging on, so that they refer, without doubt, to that one variable. For instance, **&x the address at which the variable x is stored. *ptr the contents of the variable which is pointed to by ptr.**

Consider the following c statement:

```
int x=2;
```

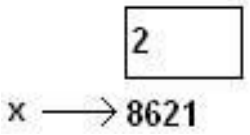
The above statement requests compiler to execute following steps:

- Reserve the required space (depending upon the data type and computer configuration) for the variable (randomly selected out of free memory pool)
- Assign as name for that reserved space
- Store the integer value 2 in the reserved space

And we can find out the starting location number of that reserve space by the following statement, & operator is used find the address of the variable.

```
printf ("Address is %u", &x);
```

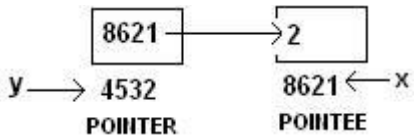
Suppose we get the output of the above statement as 8621 which is the starting address of the reserved space and hence we can represent the memory map for the variable i as:




If we run the same statement **int x=2;** again then the same steps would be following but the memory location selected can be different from 8621. Reason being that the compiler randomly selects the location from the large pool of free memory spaces and every time we select it, there are rare chances of same memory location being selected.

The pointer, as the name suggests, is something which is pointing to or directing to something. The pointer is used to store the address of another variable and the data type of the pointer is same as data type of variable it is pointing to. As the variable which points to other

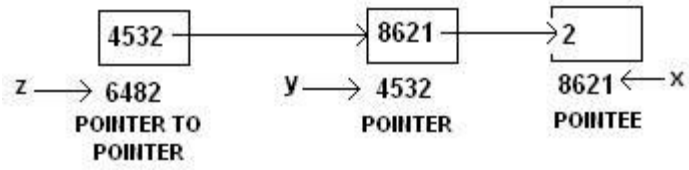
variable is called pointer and the variable which is pointed to by the pointer is called **Pointee**. We represent the relation between the two as:




We can also have pointer to pointer and we declare it as:

```
int **z;
z=&y;
```

And the memory map can be represented as:




Note: Types of operations on pointers like addition of pointers, multiplying pointer by number, dividing pointer by number are illegal.

### 1.9.3 POINTERS AND INITIALIZATION:

Something to be varying of with pointer variables is the way that they are initialized. It is incorrect, logically, to initialize pointers in a declaration. A compiler will probably not prevent this however because there is nothing incorrect about it as far as syntax is concerned. Think about what happens when the following statement is written. This statement is really talking about two different storage places in the memory:

```
int *a = 2;
```

First of all, what is declared is a pointer, so space for a `pointer to int' is allocated by the program and to start off with that space will contain garbage (random numbers), because no statement like `a = &some int;` has yet been encountered which would give it a value. It will then attempt to fill the contents of some variable, pointed to by a, with the value 2. This is doomed to failure. ‘a’ only contains garbage so the 2 could be stored anywhere. There may not even be a variable at the place in the memory which a points to. Nothing has been said about that yet. This kind of initialization cannot possibly work and will most likely crash the program or corrupt some other data.

### 1.9.4 TYPECASTING IN POINTERS:

It is tempting but incorrect to think that a pointer to an integer is the same kind of object as a pointer to a floating point object or any other type for that matter. This is not necessarily the case. Compilers distinguish between pointers to different kinds of objects. There are occasions however when it is actually necessary to convert one kind of pointer into another. This might happen with a type of variable called "unions" or even functions which allocate storage for special uses. These objects are met later on in this book. When this situation comes about, the cast operator has to be used to make sure that pointers have compatible types when they are assigned to one another. The Cast Operator, is written in front of a variable to force it to be a particular type: `(type) variable`
For pointers it is: `(type *) pointer`

Look at the following statement:

```
char *ch;
int  *i;
i = (int *) ch;
```

This copies the value of the pointer ch to the pointer i. The cast operator makes sure that the pointers are in step and not talking at cross purposes. The reason that pointers have to be 'cast' into shape is a bit subtle and depends upon particular computers. In practice it may not actually do anything, but it is a necessary part of the syntax of C.

1.9.5 POINTER EXPRESSIONS:

Like any other variable, pointer variable can be used in arithmetic expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y=*p1**p2;
sum=sum+*p1;
z=5*-*p2/p1;
*p2 = *p2 + 10;
```

C language allows us to add integers to, subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc., we can also compare pointers by using relational operators the expressions such as p1 > p2, p1==p2 and p1! =p2 are allowed.

/*Program to illustrate the pointer expression and pointer arithmetic*/

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

```
{
      int  ptr1,ptr2;
      int  a,b,x,y,z;
      a=30;b=6;
      ptr1=&a;
      ptr2=&b;
      x=*ptr1+*ptr2-6;
      y=6*-*ptr1/*ptr2+30;
      printf("nAddress    of   a    +%u",ptr1);
      printf("nAddress    of   b    %u",ptr2);
      printf("na=%d,             b=%d",a,b);
      printf("nx=%d,y=%d",x,y);
      ptr1=ptr1      +    70;
      ptr2=     ptr2;
      printf("na=%d,      b=%d",a,b);
}
```

### 1.9.6 POINTER INCREMENTS AND SCALE FACTOR:

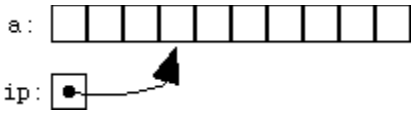
Pointers can be incremented like, `p1 = p1 + 2; p1 = p1 + 1;` an expression can be like `p1++;` Will cause the pointer `p1` to point to the next value of its type. For example, if p1 is an integer pointer with an initial value, say 2800, then after the operation `p1 = p1 +1`, the value of `p1` will be `2802`, and not `2801`. Its value is increased by the length of the data type that its points to. This length is called the scale factor.

### 1.9.7 POINTERS AND ARRAYS:

Pointers do not have to point to single variables. They can also point at the cells of an array. For example, we can write

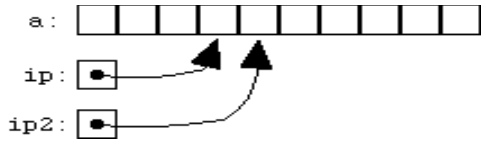
```
int *ip;
int a[10];
ip = &a[3];
```

We would end up with ip pointing at the fourth cell of the array a (remember, arrays are 0-based, so a [0] is the first cell). We could illustrate the situation like this:




We would use this ip just like the one in the previous section: *ip gives us what ip points to, which in this case will be the value in a [3]. Once we have a pointer pointing into an array, we can start doing pointer arithmetic. Given that ip is a pointer to a [3], we can add 1 to ip: ip + 1;

In C, it gives a pointer to the cell one farther on, which in this case is a[4]. To make this clear, let's assign this new pointer to another pointer variable: ip2 = ip + 1;
Now the picture looks like this:




1.9.8 ARRAYS OF POINTERS:

We can have arrays of pointers since pointers are variables. *Arrays of Pointers* are a data representation that will cope efficiently and conveniently with variable length text lines. This eliminates:

- Complicated storage management
- High overheads of moving lines

Syntax of declaring a pointer:

```
data_type_name      * variable     name
```

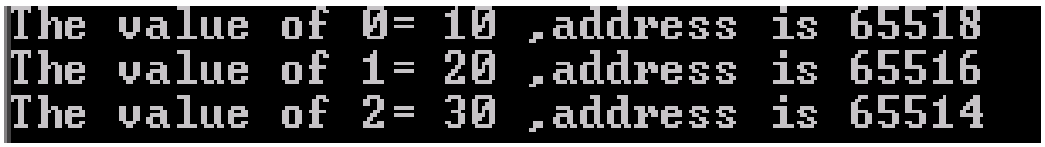
First of all, specify the data type of data stored in the location, which is to identified by the pointer. The asterisk tells the compiler that you are creating a pointer variable. Then specify the name of variable. You can see in the given example, we have created an array of pointers of maximum size 3. Then we have assigned the objects of array pointer.

```
array[0] = &x;
array[1] = &y;
array[2] = &z;
```

```
#include <stdio.h>
#include <conio.h>
main()
{
     clrscr();
     int *array[3];
     int x = 10, y = 20, z = 30;
     int i;
     array[0] = &x;
     array[1] = &y;
     array[2] = &z;
     for (i=0; i< 3; i++)
```

```
        {

            printf("The value of %d= %d ,address is %u\t \n", i,
     *(array[i]),array[i]);
        }
        getch();
        return 0;
    }
```

Output:




The information about how arrays are stored was not included just for interest. There is another way of looking at arrays which follows the BCPL idea of an array as simply a block of memory. An array can be accessed with pointers as well as with [] square brackets. The name of an array variable, standing alone, is actually a pointer to the first element in the array. For example: if an array is declared float `numbers [34];` numbers is a pointer to the first floating point number in the array; numbers is a pointer in its own right. (In this case it is type 'pointer to float'.) So the first element of the array could be accessed by writing:
numbers [0] = 22.3;
*numbers = 22.3;

1.9.9 MEMORY MAP:

In C language we have different types of variables like global variable, local variable, parameter to the function and dynamically allocated variable. But do we know how memory is allocated to these variables and where are these variables stored? This concept is very important to understand. So let's discuss this and understand the language better which would help us in dealing with many errors in the program.

TEXT SEGMENT(.text):

Text segment, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

DATA SEGMENT (DS):

The data area contains global and static variables used by the program that are initialized. This segment can be further classified into initialized read-only area and initialized read-write area. All static and extern variable are stored in the data area. It is permanent memory space and variable will store in the memory unless and until program end. Initialized data segment, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

```
int   maxcount = 99;
```

BSS SEGMENT(.bss):

The BSS segment also colloquially known as *uninitialized data* starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code. For instance a variable declared (static int i;) would be contained in the BSS segment. Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "**block started by symbol.**"

HEAP MEMORY:

The heap memory is used to allocate space to the dynamic data. The data whose size can shrink and expand as the requirements vary. We can allocate the extra memory whenever required using command such as malloc (), calloc (), realloc () and de-allocate the space already allocated using command such as free (), delete ().

The memory in heap is allocated at the run time. The heap area begins at the end of the BSS segment and grows to larger addresses. We can take HEAP memory as a group of nodes (circles) and every node is reachable only if It has a direct or indirect reference from some root node (pointer) which is stored in STACK memory. When we use memory allocation functions like malloc, calloc or realloc the memory node are selected and linked to the specified root node. Now if we don't free the nodes of HEAP memory before the root node is de-allocated, the reference to those memory nodes is lost and they become unreachable, hence known as garbage memory or memory leak. Hence to avoid this problem the programmer must free the memory after its use.

STACK MEMORY:

The stack memory is a data structure which follows the policy of Last in First out (LIFO).The stack memory is used to store the static data which includes the variables which are declared at the start of the program like the fixed arrays, local variables and the parameters which are passed to a function during the function call. Other than this the return address pointer, frame pointer (the pointer which is used to point to the starting of the frame just below)

and other registers. The life of the static variables ends as soon as the function returns or the program execution stops. The stack memory is used separately for separate programs in an operation system. The memory in stack is allocated at the compile time.
**Disadvantage**: The memory declared first remains the same i.e. if we declare large memory then there is wastage of memory while if we declare lesser memory then we are in a problem.
**Advantage**: The memory allocated is de allocated automatically after the function returns and hence the same memory is available for re-use again

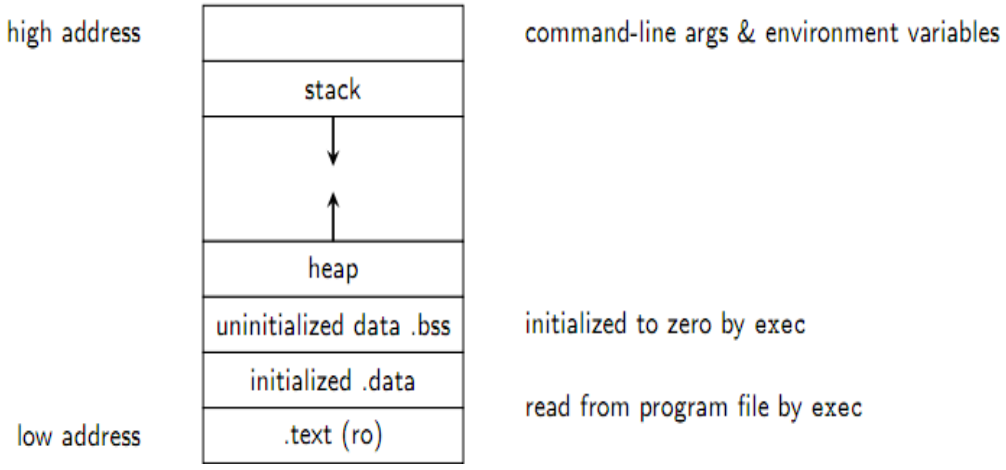



**Figure: Memory management in C**

**Advantage:** This memory can be anytime shrunk or expanded whenever the requirements of the program changes.

**Disadvantage:** We have to manually de allocate the memory allocated using function like free or delete, otherwise the memory which is not de allocated at the proper time goes to the garbage and is not available for re-use.
Illustration with Simple Program:

```
int x=4;
main()
{
      int z;
      char* y;
      y= malloc(6);
      y="hello";
      printf("%c", *y);
      z = square(x);
```

```
        printf("%d", z);
        free(y);
}

int square(int k)
{
        return k*k;
}
```

Now we see which variables in the program are stored where. In this program we have different types of variables like x as global variable, z & k as local variable and as parameter to the function and y is dynamically allocated variable. So x is stored in fixed memory, z & k are stored in Stack Memory and the pointer y is also stored in stack memory while the 6 bytes (allocated by malloc) which are pointees of pointer y are stored in heap memory.

## 1.10: POINTERS AND FUNCTIONS

### 1.10.1 POINTERS AS FUNCTION ARGUMENTS:

When an array is passed to a function as an argument, only the address of first element of the array is passed, but not the actual values of the array elements. If x is an array, when we call sort(x), the address of x [0] is passed to the function sort (). The function uses this address for manipulating the array elements. Similarly we can pass the address of a variable as an argument to a function in the normal fashion.

When we pass address to a function, the parameters receiving the address should be pointers. The process of calling a function using pointers to pass the addresses of variable is known as *call by reference*. The function which is called by 'reference' can change the value of the variable used in the call. Consider the following code:

```
main()
{
        int x;
        x=20;
        change (&x);
        printf ("%d\n", x);
}
change (p)
        int *p;
        {
```

```
                    *p = *p +10;
                    }
}
```

When the function change () is called, the address of the variable x, not its value, is passed into the function change (). Inside change (), the variable p is declared as a pointer and therefore p is the address of the variable x.

```
*p = *p + 10;
```

By this statement, x value is changed to 30.

1.10.2 FUNCTIONS RETURNING POINTERS:

Any function can return pointer values. Return values must have same data type as function.

```
int *fun();
void main()
{
     int *p;
     p = fun();
     printf("the value of p = %u \n", p);
     printf("the value of *p = %u\n", *p);
     getch();
}
int *fun()
{
     static int i=20;
     return(&i);
}
```

Output:
The value of p = 3000
The value of *p = 20

1.10.3 POINTER TO FUNCTIONS:

A function, like a variable, has an address location in memory. So, it is possible to declare a pointer to function, which can then be used as an argument in another function. A pointer to function is declared as:

```
Data type (fptr) ();
```

This tells the complier that fptr is a pointer to a function which returns type value. The parenthesis around *fptr are necessary.

For statement like type *gptr (); would declare gptr as a function returning a pointer to *type*.We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer, for example

```
double (*p1)(), mul();
p1 = mul;
```

Declare p1 as a pointer to function and mul as a function and then make p1 to point to the function mul. To call the function mul, we have to know the list of parameters,

```
                    (*p1)(x ,y)
Is equivalent to    mul(x, y)
```

### 1.10.4 POINTER TO POINTER:

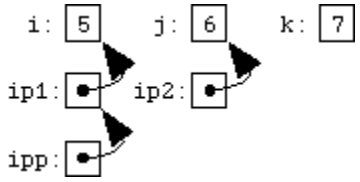
Since we can have pointers to int, and pointers to char, and pointers to any structures we've defined, and in fact pointers to any type in C, it shouldn't come as too much of a surprise that we can have pointers to other pointers. If we're used to thinking about simple pointers, and to keeping clear in our minds the distinction between *the pointer itself* and *what it points to*, we should be able to think about pointers to pointers, too, although we'll now have to distinguish between the pointer, what it points to, and what the pointer that it points to points to. (And, of course, we might also end up with pointers to pointers to pointers, or pointers to pointers to pointers to pointers, although these rapidly become too esoteric to have any practical use.) ,The declaration of a pointer-to-pointer looks like

```
int **ipp;
```

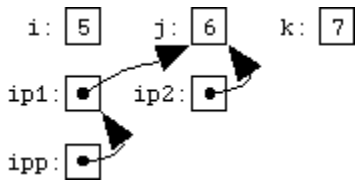
Where the two asterisks indicate that two levels of pointers are involved.
Starting off with the familiar, uninspiring, kindergarten-style examples, we can demonstrate the use of ipp by declaring some pointers for it to point to and some ints for those pointers to point to:

```
int i = 5, j = 6; k = 7;
int *ip1 = &i, *ip2 = &j;
```

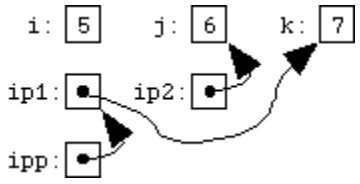
Now we can set `ipp = &ip1;` and ipp points to ip1 which points to i. *ipp is ip1, and **ipp is i, or 5. We can illustrate the situation, with our familiar box-and-arrow notation, like this:



If we say `*ipp = ip2;` we've changed the pointer pointed to by ipp (that is, ip1) to contain a copy of ip2, so that it (ip1) now points at j:




If we say `*ipp = &k;` we've changed the pointer pointed to by ipp (that is, ip1 again) to point to k:




What are pointers to pointers good for, in practice? One use is returning pointers from functions, via pointer arguments rather than as the formal return value. To explain this, let's first step back and consider the case of returning a simple type, such as int, from a function via a pointer argument. If we write the function

```
f(int *ip)
{
    *ip = 5;
}
```

and then call it like this:

```
int i;
f(&i);
```

Then f will ``return'' the value 5 by writing it to the location specified by the pointer passed by the caller; in this case, to the caller's variable i. A function might ``return'' values in this way if it had multiple things to return, since a function can only have one formal return value (that is, it can only return one value via the return statement.) The important thing to notice is that for the function to return a value of type int, it used a parameter of type pointer-to-int.

Example Program:

```
void main ()
{
    int i = 3;
    int *j;
    int **k;
    j = &i;
```

```
        k = &j;
        printf("the address of i = %u\n", &i);
        printf("the address of i = %u\n", j);
        printf("the address of j = %u\n", k);
        printf("the address of j = %u\n", &j);
        printf("the address of k = %u\n", &k);
        printf("the address of j = %u\n", j);
        printf("the address of j = %u\n", *(&i));
        printf("the address of j = %u\n", **k);
}
```

Output:
the address of I = 400
the address of I = 4000
the address of j = 5000
the address of j = 5000
the address of k = 5000
the address of j = 5000
the address of j = 3
the address of j = 3

### 1.10.5 POINTER DRAWBACKS:

Pointers have tremendous power but the power can swing both sides good and evil. Pointer if used incorrectly leads to very difficult to unearth bugs which will most probably make you go wild. Pointers are itself are not any trouble but the value they are storing can be. If it contains a incorrect value it can lead to disasters of massive magnitude when used.

When you use this incorrect pointer to read a memory location, you may be reading a incorrect garbage value which if unluckily accepted by your program as assumed correct value nothing can help you. Consider a scenario in banking in which any customers real account value is switched with this garbage value, he can become a millionaire or beggar in a second, or think that in a rocket launching software you use this incorrect value as launching angle and crashing the billion dollar masterpiece. These scenarios are just my imagination running wild but you cannot ignore the fact that they are possibility.

Now when you use this incorrect pointer to write a memory location you may be writing a unknown memory location. If you have a large memory in the system maybe you are using a unassigned memory but if that memory by any luck is a memory used by O.S. or Hardware and you are modifying it you maybe corrupting your Operating System software or damaging your hardware and their drivers. Also it is a possibility that you may be using a memory

location already in use by your software storing some essential data and you are unknowingly modifying it. You may be writing over your own code and data.

To help you avoid these pointers error, we will be looking at some types of pointers error which are possible and learn how to avoid them by using good programming practices.

Uninitialized Pointers

Carefully look at the c source code below –

```
#include <stdio.h>
int main ()
{
 int a, *p;
  a = 1;
 *p = a;
  return 0;
}
```

Now notice above that pointer p above is pointing to some unknown location. Maybe if you are luckily your compiler or O.S. Points it to some safe location but it is just an assumption. It can be pointing to any damn memory location in the system. Now the code above write value 1 into the unknown memory location pointed to by p. There is a large possibility with large program in small environment that p is pointing something vital which is now destroyed. To avoid this error you must use pointer initialization so that pointer is pointing to nothing and compiler issue an error (runtime or compile time) when you use that pointer incorrectly and some damage control can happen.

Correct and Safe Code would be –

```
#include <stdio.h>
int main ()
{
      int a, *p = NULLL;
      a = 1;
       *p = a; //This would generate a run time error
      always and a compile time error in
            smart compilers.
      return 0;
}
```

Incorrect Pointer Usage

Carefully look at the c source code below -

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

```
#include <alloc.h>
int main ()
{
      int a, *p = NULLL;
      p = (int *) malloc (size of (int));
      a = 1;
      p = a;
      printf ("%d",*p); //This will print garbage value
      return 0;
}
```

Seeing above code i guess you can easily point the error. Line p = a; is incorrect and should be p = &a; the point is this maybe a syntax error by the programmer or a logical error by a beginner it can be disastrous. Compiler should give warning but no compiler would produce error, in fact even a warning is only a assumption and may not occur. The reason being compiler except this because every point in memory is represented in hexa decimal number which this decimal number is implicitly converted. It is a rare but useful thing in special circumstances to use hardcode memory values especially in case of hardware programming and system programming where values of memory address such as port address always remain constant for the machine for which the program is being used. But this concept allows accidental error like above which may prove dangerous. So stay cautious of such error while programming.

Correct and Safe Code would be -

```
#include <stdio.h>
#include <alloc.h>
int main ()
{
      int a, *p = NULLL;
      p = (int *) malloc (sizeof (int));
       a = 1;
       p = &a;
       prinf ("%d",*p);
       return 0;
}
```

Incorrect Pointer Comparison

Carefully look at the c source code below -

```
#include <stdio.h>
#include <alloc.h>
int main ()
```

```
{
        int *p = NULLL, *q = NULLL;
        p = (int *) malloc (size of (int));
        q = (int *) malloc (sizeof (int));
       if ( p < q )
       printf ("\np is less than q"); //This will print
                                        garbage value
        else
        printf ("\np is greater than q");
        return 0;
}
```

The comparison of pointer as done above is absolutely incorrect reason being that both p and q pointers in code above are entirely unrelated and the output of the above code cannot be predicted. Comparison done above is entirely dependent upon compiler, operating system and to great extent luck. When pointer comparison is performed, the comparison is done between the integer values of the memory locations they are pointing to. Now this memory location assigned to them is not fixed and is entirely random so p can be either less or greater than q each with 50% probability in mathematics terms. So the comparison cannot be relied upon is the bottom line. But this comparison if used correctly can be useful in logically related memory allocation such as in arrays. Also in very rare conditions you may even want to use this unrelated pointer comparison.

## 1.11 STRUCTURES IN C

1.11.1: INTRODUCTION:

We have seen that arrays can be used to represent a group of data items that belong to the same type, such as int or float. However, if we want to represent a collection of data items of different types using a single name, then we cannot use an array. C supports a constructed data type known as structure, which is a method for packing data for different data types. A structure is a convenient took for handling a group of logically related data items. A structure is a set of attributes, such as student name, rool_number and marks. The concept of a structure is analogous to that of a record in many other languages. Structures help to organize in a more meaningful way. It is a powerful concept that we may often need to use in our program design.

1.11.2 STRUCTURE DEFINITION:

The general format of a structure definition is as follows:

```
struct tag_name
{
     data type member1;
     data type member1;
};
```

A structure definition creates a format that may be used to declare structure variables. Consider a structure as follows:

```
struct  book_bank
{
     char title[20];
     char author[15];
     int pages;
     float price;
};
```

The keyword struct declares a structure to hold the details of four fields, namely title, author, pages and price. These fields are called structure elements or members. Each member may belong to a different type of data. Book_bank is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have tag's structure.

We can declare structure variables using the tag name anywhere in the program, for example the statement **struct book_bank book1, book2, book3;** Declares **book1, book2, book3** as variables of type **struct book_bank.** Each one of these has four variables as specified by the template. The complete declaration might look like this:

```
struct book_bank
{
     char title[20];
     char author[15];
     int pages;
     float price;
};
Struct book_bank book1, book2, book3;
```

Remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as **book1.**

In defining a structure we have note the following syntax:

- The template is terminated by semicolon.
- While the entire declaration is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
- The tag name such as book_bank can be used to declare structure variables of its type, later in the program.

The use of tag name is optional. For example:

```
struct
{
……
}book1, book2, book3;
```

Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the main, along with macros definitions, such as #define. In such cases, the definition is global and can be used by other functions as well.

### 1.11.3 ASSIGNING VALUES:

We can assign values to the members of a structure in a number of ways. The members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word title has no meaning where as the phrase 'title of book3' has a meaning. The link between a member and a variable is established using the member operator '.' Which is also known as 'dot operator' or 'period operator'.
For example, `book1.price` is the variable representing the price of book1 and can be treated like any other ordinary variable.

```
strcpy(book1.title, "BASIC");
strcpy(book1.author, "dennis");
book1.pages = 250;
book1.price = 28.50;
```

### 1.11.4 STRUCTURE INITIALIZATION:

Like any other data type, a structure variable can be initialized. A structure must be declared as static if it is to be initialized inside a function (similar to arrays). This condition is not applicable to ANSI compliers. The ANSI compliers permit initialization of structure variables with auto storage class. Structure initialization is as follows:

```
main()
{
      struct st_record
      {
            int weight;
            float height;
      };
      static struct st_record student1 = {60,180.75};
      static struct st_record student2 = {53,170.60};
      ……..
      ……..
}
```

Another method for declaration:

```
struct st_record
{
      int weight;
      float height;
} student1 = {60,180.75};
main()
{
      Static struct st_record student2 ={53, 170.60};
}
```

1.11.5 COMPARISON OF STRUCTURE VARIABLES:

Two variables of structure members can be compared as original variables. If **person1** and **person2** are of same structure, then the following operations are valid:

| Operation | Meaning |
|---|---|
| person1 = person2 | Assign person2 to person1 |
| person1 == person2 | Compare all members of person1 and person2 and return 1 if they are equal, 0 otherwise. |
| person1 != personm2 | Returns 1 if all the members are not equal, 0 otherwise. |

1.11.6 ARRAY OF STRUCTURES:

We use structures to describe the format of a number of related variables. For example for analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example, **struct class student [100];** Defines an array called student that consists of 100 elements. Each element is defined to be of type **struct class.**

Consider the following declaration:

```
struct marks
{
      int subject1;
      int subject2;
      int subject3;
};
main()
{
      static struct marks student[3]={{45,68,81}, {75,53,69},{57,36,71}};
}
```

This declares the **student** as an array of three elements **student[0]**, **student[1]**, **student[2]** and initializes their members as follows:

```
student[0].subject1 = 45;
student[1].subject1 = 68;
student[2].subject1 = 57;
```

Note that the array is declared just as it would have been, with any other array. Since student is an array, we use the usual array- accessing methods to access individual elements and then the member operator to access members. An array of structures is stored inside the memory in the same way as a multi-dimensional array.

```
student[0].subject1    =   45
student[0].subject2    =   68
student[0].subject3    =   81
student[1].subject1    =   75
student[1].subject2    =   53
student[1].subject3    =   69
student[2].subject1    =   57
```

```
student[2].subject2    =   36
student[2].subject3    =   71
```

Illustration program for array of structures:

```
struct marks
{
        int sub1;
        int sub2;
        int sub3;
        int total;
};
main()
{
        int i;
        static struct marks student[3]={{45,67,81,0},{75,53,69,0}, {57,36,71,0}};
        static struct marks total;
        for(i=0; i<=2; i++)
        {
                student[i].total = student[i].sub1 + student[i].sub2 + student[i].sub3;
                total.sub1 = total.sub1 + student[i].sub1;
                total.sub2 = total.sub2 + student[i].sub2;
                total.sub3 = total.sub3 + student[i].sub3;
                total. total = total. total + students[i].total;
        }
        printf("student total \n\n");
        for(i=0; i<=2; i++)
        printf("student[%d]  %d\n", i+1, student[i].total);
        printf("grand total = %d\n", total.total);
}
```

Output:

```
STUDENT     TOTAL
Student [1]      193
Student [2]      197
Student [3]      164
GRAND TOTAL = 554
```

1.11.7 ARRAYS WITHIN STRUCTURES:

C permits the use of arrays as structure members. we have already used array of characters inside a structure. Similarly, we can use single or multi-dimensional arrays of type int or float. For example the following structure declaration is valid:

```
struct marks
{
     int number;
     float subject[3];
}student[2];
```

Here, the member **subject** contains three elements, **subject[0], subject[1] and subject[2].** These elements can be accessed using appropriate subscripts. For example, the name: **student[1].subject[2];** would refer to the marks obtained in the third subject by the second student.

### 1.11.8 STRUCTURES WITHIN STRUCTURES:

Structures within a structure mean nesting of structures. Nesting of structures is permitted in c. let us consider the following structure defined to store information about the salary of employees.

```
struct salary
{
     char name[20];
     char department[10];
     int basic_pay;
     int dearness_allowance;
     int house_rent_allowance;
     int city_allowance;
} employee;
```

This structure defines names, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a substructure as:

```
struct salary
{
     char name[20];
     char department[10];
     struct
     {
```

```
            int dearness;
            int house_rent;
        I   nt city;
        } allowance;
}employee;
```

The salary structure contains a member names **allowance** which itself is a structure with three members. The members contained in the inner structure namely **dearness**, **house_rent**, and **city** can be referred to as

```
employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city
```

An inner structure in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator. The following being invalid:

```
employee.allowance   (actual member is missing)
employee.house_rent   (inner structure variable is missing)
```

An inner structure can have more than one variable.

```
struct salary
{
struct
{
 int dearnesss;
}
allowances;
arrears;
}employees[100];
```

The inner structure has two variables, **allowances** and **arrears.** This implies that both of them have the same structure template. A base member can be accessed as follows:

```
employee[1].allowance.dearness;
employee[1].arrears.allowances;
```

We can also use tag names to define inner structures. Example:

```
struct pay
{
     int dearness;
```

```
        int house_rent;
        int city;
};
        struct salary
        {
              char name[20];
              char department[10];
              struct pay allowance;
              struct pay arears;
        }struct salary employee[100];
```

It is also permissible to nest more than one type of structures:

```
        struct personal_record
        {
              struct name_part name;
              struct addr_part address;
              struct date date_of_birth;
        }struct personal_record person1;
```

1.11.9 STRUCTURES AND FUNCTIONS:

C supports the passing of structures values as arguments to functions. There are there methods by which values can be transferred from one function to another function. The first method is to pass each member of the structure as the actual argument of the function call. The actual arguments can be treated independently as ordinary variables. This is the most elementary method and unmanageable when the structure size is large.

The second method involves passing of a copy of the structure to the calling function. Since function is working on a copy of the structure, any changes to the structure members within the function are not reflected to the original structure (in the calling function). It is therefore, necessary for the function to return the entire structure back to the calling function.

The third approach employs a concept called pointers to pass the structure as an argument. In this case, the address location of the structure is passed to the function. The function can indirectly access the entire function and work on it. This method is more efficient.

The general format of sending a copy of a structure to the called function is:

```
function name (structure variable name)
```

The following points are important to note:

- The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as **struct** with an appropriate tag name.
- The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
- The **return** statement is necessary only when the function is returning some data.
- When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
- The called function must be declared in the calling function for its type, if it is placed after the calling function.

### 1.11.10 POINTERS TO STRUCTURES:

We know that the name of an array stands for the address of its zeroth element. The same thing is true for the names of arrays of structure variables. Suppose product is an array variable of struct type. The name product represents the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
char name[30];
int number;
float price;
}product[2], *ptr;
```

This statement declares product as an array of two elements, each of the type struct inventory and ptr as an pointer to data objects of the type struct inventory.

```
ptr = product;
```

The assignment would assign the address of the zeroth element of product to ptr. That is, the pointer ptr will now point to product[0]. Its members can be accessed using following notation.

```
ptr → name
ptr → number
ptr → price
```

The symbol ( → ) is called arrow operator and is made up of a minus sign and a greater than sign. Note that ptr -> is simply another way of writing product[0]. When the pointer is incremented by one, it is made to point to the next record, i.e., product[1]. The following for statement will print the values of members of all the elements of product array.

```
for(ptr = product; ptr < product+2; ptr++)
```

```
printf("%s %d %f \n",ptr ->name, ptr -> number, ptr -> price);
```

We could also use the notation to access the member number. The parenthesis around *ptr are necessary because the member operator "." Has higher precedence than the operator *.

```
(*ptr).number
```

While using structure pointers, we should take care of the precedence of operators.

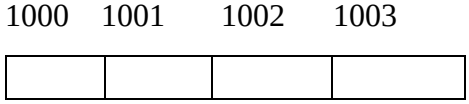
1.11.11 UNIONS:

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is a major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of the union use the same location. Thos implies that, although a union may contain members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword union as follows:

```
union item
{
     int m;
     float x;
     char c;
}code;
```

This declares a variable code of type union item. The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

Storage of 4 bytes




The complier allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above the member x requires 4 bytes which is the largest among the members. To access the union members, we can use the same syntax that we use for structure members.

```
code.m
code.x
code.c
```

In effect, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the

previous member’s value. Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

# 1.12: STRUCTURE MEMBER ALIGNMENT, PADDING AND DATA PACKING

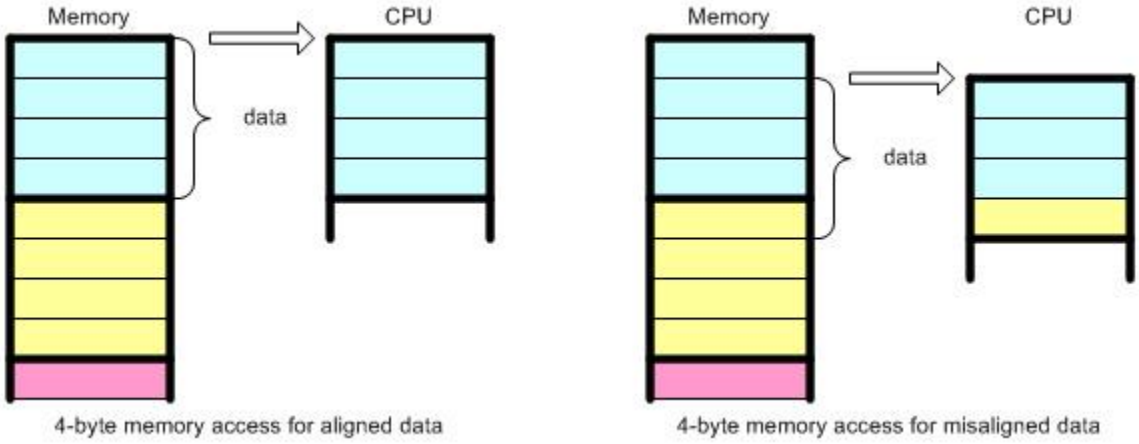
## 1.12.1 DATA ALIGNMENT:

**Data structure alignment** is the way data is arranged and accessed in computer memory. It consists of two separate but related issues: *data alignment* and *data structure padding*. When a modern computer reads from or writes to a memory address, it will do this in word sized chunks (e.g. 4 byte chunks on a 32-bit system). *Data alignment* means putting the data at a memory offset equal to some multiple of the word size, which increases the system's performance due to the way the CPU handles memory. To align the data, it may be necessary to insert some meaningless bytes between the end of the last data structure.

For example, when the computer's word size is 4 bytes, the data to be read should be at a memory offset which is some multiple of 4. When this is not the case, e.g. the data starts at the 14th byte instead of the 16th byte, and then the computer has to read two 4-byte chunks and do some calculation before the requested data has been read, or it may generate an alignment fault. Even though the previous data structure ends at the 14th byte, the next data structure should start at the 16th byte. Two padding bytes are inserted between the two data structures to align the next data structure to the 16th byte. Although data structure alignment is a fundamental issue for all modern computers, many computer languages and computer language implementations handle data alignment automatically.

In programming language, a data object (variable) has 2 properties; its value and the storage location (address). Data alignment means that the address of a data can be evenly divisible by 1, 2, 4, or 8. In other words, data object can have 1-byte, 2-byte, 4-byte, 8-byte alignment or any power of 2. For instance, if the address of a data is 12FEECh (1244908 in decimal), then it is 4-byte alignment because the address can be evenly divisible by 4. (You can divide it by 2 or 1, but 4 is the highest number that is divisible evenly.)
CPU does not read from or write to memory one byte at a time. Instead, CPU accesses memory in 2, 4, 8, 16, or 32 byte chunks at a time. The reason for doing this is the performance - accessing an address on 4-byte or 16-byte boundary is a lot faster than accessing an address on 1-byte boundary.

The following diagram illustrates how CPU accesses a 4-byte chuck of data with 4-byte memory access granularity.

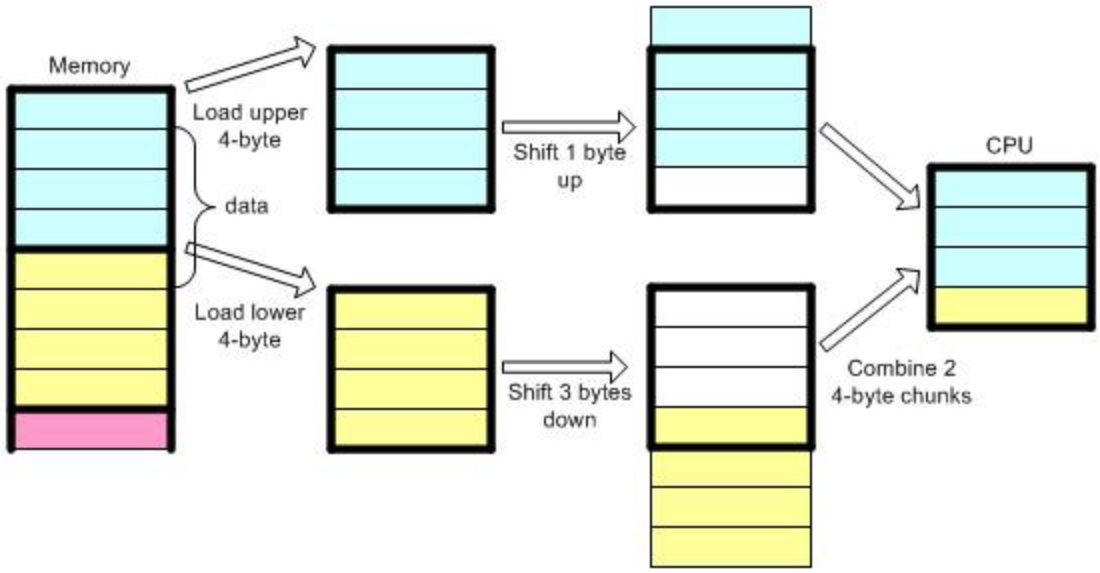



Example Program:

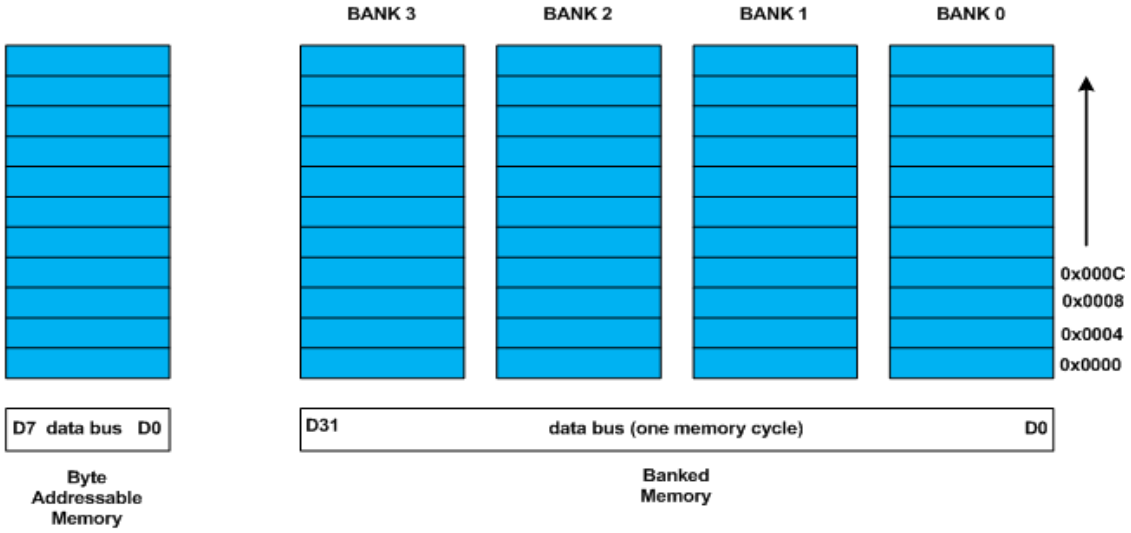
```
#include <stdio.h>

// Alignment requirements
// (typical 32 bit machine)

// char         1 byte
// short int    2 bytes
// int          4 bytes
// double       8 bytes

// structure A
typedef struct structa_tag
{
   char    c;
   short int  s;
} structa_t;

// structure B
typedef struct structb_tag
{
   short int  s;
   char    c;
   int     i;
} structb_t;
```

```
// structure C
typedef struct structc_tag
{
   char    c;
   double   d;
   int    s;
} structc_t;

// structure D
typedef struct structd_tag
{
   double   d;
   int    s;
   char    c;
} structd_t;

int main()
{
   printf("sizeof(structa_t) = %d\n", sizeof(structa_t));
   printf("sizeof(structb_t) = %d\n", sizeof(structb_t));
   printf("sizeof(structc_t) = %d\n", sizeof(structc_t));
   printf("sizeof(structd_t) = %d\n", sizeof(structd_t));

   return 0;
}
```

If the data is misaligned of 4-byte boundary, CPU has to perform extra work to access the data: load 2 chucks of data, shift out unwanted bytes then combine them together. This process definitely slows down the performance and wastes CPU cycle just to get right data from memory.

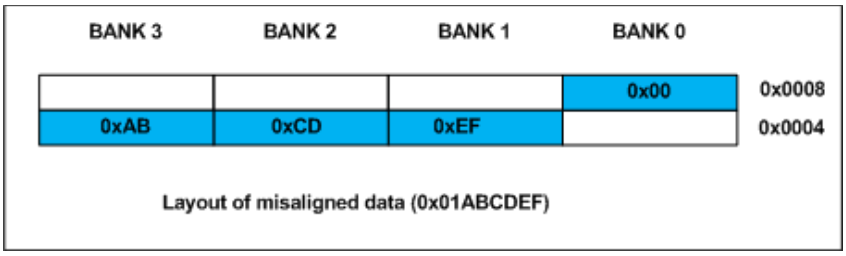



Every data type in C/C++ will have alignment requirement (in fact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.




Historically memory is byte addressable and arranged sequentially. If the memory is arranged as single bank of one byte width, the processor needs to issue 4 memory read cycles to fetch an integer. It is more economical to read all 4 bytes of integer in one memory cycle. To take such advantage, the memory will be arranged as group of 4 banks as shown in the above figure. The memory addressing still be sequential. If bank 0 occupies an address X, bank 1, bank 2 and bank 3 will be at (X + 1), (X + 2) and (X + 3) addresses. If an integer of 4 bytes is allocated on X address (X is multiple of 4), the processor needs only one memory cycle to read entire integer. Whereas, if the integer is allocated at an address other than multiple of 4,

it spans across two rows of the banks as shown in the below figure. Such an integer requires two memory read cycle to fetch the data.




A variable's data alignment deals with the way the data stored in these banks. For example, the natural alignment of int on 32-bit machine is 4 bytes. When a data type is naturally aligned, the CPU fetches it in minimum read cycles.
Similarly, the natural alignment of ***short int*** is 2 bytes. It means, a ***short int*** can be stored in bank 0 – bank 1 pair or bank 2 – bank 3 pair. A ***double*** requires 8 bytes, and occupies two rows in the memory banks. Any misalignment of ***double*** will force more than two read cycles to fetch ***double*** data.

1.12.2 STRUCTURE PADDING:

In C/C++ a structures are used as data pack. It doesn't provide any data encapsulation or data hiding features (C++ case is an exception due to its semantic similarity with classes). Because of the alignment requirements of various data types, every member of structure should be naturally aligned. The members of structure allocated sequentially increasing order. Let us analyze each struct declared in the above program.

OUTPUT OF ABOVE PROGRAM:
**Structure A:**

The *structa_t* first element is *char* which is one byte aligned, followed by *short int*. short int is 2 byte aligned. If the the short int element is immediately allocated after the char element, it will start at an odd address boundary. The compiler will insert a padding byte after the char to ensure short int will have an address multiple of 2 (i.e. 2 byte aligned). The total size of structa_t will be sizeof(char) + 1 (padding) + sizeof(short), 1 + 1 + 2 = 4 bytes.

**Structure B:**

The first member of *structb_t* is short int followed by char. Since char can be on any byte boundary no padding required in between short int and char, on total they occupy 3 bytes. The next member is int. If the int is allocated immediately, it will start at an odd byte boundary. We need 1 byte padding after the char member to make the address of next int member is 4 byte aligned. On total, the *structb_t* requires 2 + 1 + 1 (padding) + 4 = 8 bytes.

**structure C – Every structure will also have alignment requirements**

Applying same analysis, *structc_t* needs sizeof(char) + 7 byte padding + sizeof(double) + sizeof(int) = 1 + 7 + 8 + 4 = 20 bytes. However, the sizeof(structc_t) will be 24 bytes. It is because, along with structure members, structure type variables will also have natural alignment.

**structure D - How to Reduce Padding?**

By now, it may be clear that padding is unavoidable. There is a way to minimize padding. The programmer should declare the structure members in their increasing/decreasing order of size. An example is structd_t given in our code, whose size is 16 bytes in lieu of 24 bytes of structc_t.

1.12.3 STRUCTURE PACKING:

Sometimes it is mandatory to avoid padded bytes among the members of structure. For example, reading contents of ELF file header or BMP or JPEG file header. We need to define a structure similar to that of the header layout and map it. However, care should be exercised in accessing such members. Typically reading byte by byte is an option to avoid misaligned exceptions. There will be hit on performance. Most of the compilers provide non standard extensions to switch off the default padding like **pragmas** or command line switches.

C++, support the same notation, where *N* is a small power of two which specifies the new alignment in bytes:

- #pragma pack(*N*) simply sets the new alignment.
- #pragma pack() sets the alignment to the one that was in effect when compilation started.
- #pragma pack(push[,N]) pushes the current alignment setting on an internal stack and then optionally sets the new alignment.
- #pragma pack(pop) restores the alignment setting to the one saved at the top of the internal stack (and removes that stack entry).

GCC will generate code using the first structure packing above if one includes the following line in the code before the structure is declared:

```
#pragma pack(1)
```

GCC will use the second struct packing if the following pragma is used:

```
#pragma pack(4)
```

In my situation, the problem was that a header file felt the need to change the struct packing without changing it at the end of the header, and not all of my source files were including the offending header. I filed a bug report and protected myself from the header with this code:

```
#pragma pack(push)
#include "evil-header.h"
#pragma pack(pop)
```

1.12.4: SPECIFIC STRUCTURE PACKING WHEN USING THE GNU C COMPILER:

The GNU C compiler does not support the #pragma directives. In particular it does not support the "#pragma pack" directive. So when using the GNU C compiler, you can ensure structure packing in one of two ways

- Define the structure appropriately so that it is intrinsically packed. This is hard to do and requires an understanding of how the compiler behaves with respect to alignment on the target machine. Also it is hard to maintain.
- Use the "packed" attribute against the members of a structure. This attribute mechanism is an extension to the GNU C compiler. An example of how you would do this is below.

```
struct test
{
unsigned char  field1 __attribute__((__packed__));
unsigned short field2 __attribute__((__packed__));
unsigned long  field3 __attribute__((__packed__));
} var1, var2;
```

Note the use of the keyword "__attribute__" with the attribute "__packed__" within the double brackets (before the terminating semicolon of each member variable declaration).
An alternate way of doing the above is as below.

```
struct test
{
    unsigned char  field1;
    unsigned short field2;
    unsigned long  field3;
} __attribute__((__packed__));
typedef struct test test_t;
test_t var1, var2;
```

This will ensure that all members of the structure are packed. Note that this doesn't seem to work right if you try to combine the typedef and the struct definition or if you combine variable declarations with the structure definition.

# 1.13 ENDIAN CONCEPT AND ITS RELEVANCE IN EMBEDDED

1.13.1 INTRODUCTION:

Endianness describes how multi-byte data is represented by a computer system and is dictated by the CPU architecture of the system. Unfortunately not all computer systems are designed with the same Endian-architecture. The difference in Endian-architecture is an issue when software or data is shared between computer systems. An analysis of the computer system and its interfaces will determine the requirements of the Endian implementation of the software.

1.13.2 DEFINITION:

Endianness is the format to how multi-byte data is stored in computer memory. It describes the location of the most significant byte (MSB) and least significant byte (LSB) of an address in memory.

1.13.3 TYPES:

**Little-endian:** increasing numeric significance with increasing memory addresses

- Little-endian architectures store the least significant part first (in the lowest memory location)
- In byte architectures also known as Least Significant Byte (LSB)
- They include the x86 lines of processors (Intel, AMD)

**Big-endian:** decreasing significance; the most-significant byte first.

- Big-endian architectures store the most significant part first (in the lowest memory location).
- In byte architectures also known as Most Significant Byte (MSB)
- Includes the Motorola line of processors (e.g. pre-Intel Macintosh)

**Bi-endian:** Some architecture designs are **bi-endian** - allow handling of both endiannesses. In some cases, this is handled by transparent hardware conversion, meaning that the non-native order is a smidge slower than the native order as the (very simple, but still present) conversion has to happen. In rare cases, the hardware will have real, equal support for both, to avoid a speed hit when something of the non-hardware-natural endianness is used.

A separate issue comes in with hardware. For example, when Apple added PCI slots, most graphics cards wouldn't actually work in them because Apple basically ignored the endianness of the PC card’s implementations. The only cards that would work were those that were implemented to support Macs. (Whether this was strategic or stupid is an arguable point)

**Mixed Endianness:** It can also be said to exist - although this is not a strictly defined or agreed-on term. It usually describes architectures which deal with non-byte-sized units in memory addressing. For example, storing a 32-bit int in a 16-bit-word architecture could lead to 0x11 0x22 0x33 0x44 - or 0x33 0x44 0x11 0x22, depending on architectural details.

Integers or single-precision floating point numbers are all 32-bits long. But since each memory address can store a single byte and not four bytes, let's split the 32-bit quantity into four bytes. For example, suppose you have a 32-bit quantity written as 12345678, which is hexadecimal. Since each hex digit is four bits, we need eight hex digits to represent the 32-bit value. The four bytes are: 12, 34, 56, and 78. There are two ways to store this in memory, as shown below.

*Big-endian*: Store the most significant byte in the smallest address, as follows:

| **Address** | **Value** |
|---|---|
| 1000 | 12 |
| 1001 | 34 |
| 1002 | 56 |
| 1003 | 78 |

*Little-endian*: Store the least significant byte in the smallest address, as follows:

| **Address** | **Value** |
|---|---|
| 1000 | 78 |
| 1001 | 56 |
| 1002 | 34 |
| 1003 | 12 |

### 1.13.4 IMPORTANCE OF ENDIANNESS:

Endianness is the attribute of a system that indicates whether integers are represented from left to right or right to left. In today's world of virtual machines and gigahertz processors. Unfortunately, Endianness must be chosen every time hardware or software architecture is designed. There isn't much in the way of natural law to help decide, so implementations vary. All processors must be designated as either big-endian or little-endian. For example, the 80x86 processors from Intel and their clones are little-endian, while Sun's SPARC, Motorola's 68K, and the PowerPC families are all big-endian.

Suppose we are storing integer values to a file, and we send the file to a machine that uses the opposite endianness as it reads in the value. This causes problems because of Endianness; we'll read in reversed values that won't make sense. Endianness is also a big issue

when sending numbers over the network. Again, if we send a value from a machine of one endianness to a machine of the opposite Endianness, we'll have problems. This is even worse over the network because we might not be able to determine the endianness of the machine that sent you the data.

1.13.5 WHEN ENDIANNESS AFFECTS CODE:

Endianness doesn't apply to everything. If we do bitwise or bit shift operations on an int, we don't notice the Endianness. The machine arranges the multiple bytes, so the least significant byte is still the least significant byte, and the most significant byte is still the most significant byte. Endianness *does* matter when you use a type cast that depends on a certain endian being in use.

TO KNOW THE WHETHER THE SYSTEM IS LITTLE OR BIG ENDIAN:

```
int main()
    {
       int i = 0xf00e;
       char *ptr = (char *)&i;
       int addr1, addr2;
       unsigned char c;
       addr1 = ptr;
       ptr++;
       addr2 = ptr;
       c = (char)i;
       printf("\n i = %x \n", i);
       printf("\nlowbyte = %x higher byte = %x\n", addr1,
addr2);
       printf("\n %x", c);
    }
```

PROGRAM TO CONVERTLITTLE ENDIAN TO BIG ENDIAN:

```
       int ltobig(int i)
         {
           int j;
          j =((i>>8)&0x00ff) + ((i<<8)&0xff00);
           return j;
          }
      main()
         {
           int l = 0xabcd;
           printf("result is == %x \n\n", ltobig(l));
         }
```