# Unit 2

## 2.1 Introduction

C is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at AT&T Bell Labs. Since its creation in 1972, it's been used for a wide variety of programs including general purpose and business applications, gaming, operating systems, databases, device driver programming, graphics programming, firmware for micro-controllers, compilers/assemblers of other programming languages and so on. C is the most widely used language in the world and has a large developer community because of which it is very stable and robust.

*Q 1. Write short note on C language?*

*Q 2. Enumerate various applications of C language?*

## 2.2 History of C language

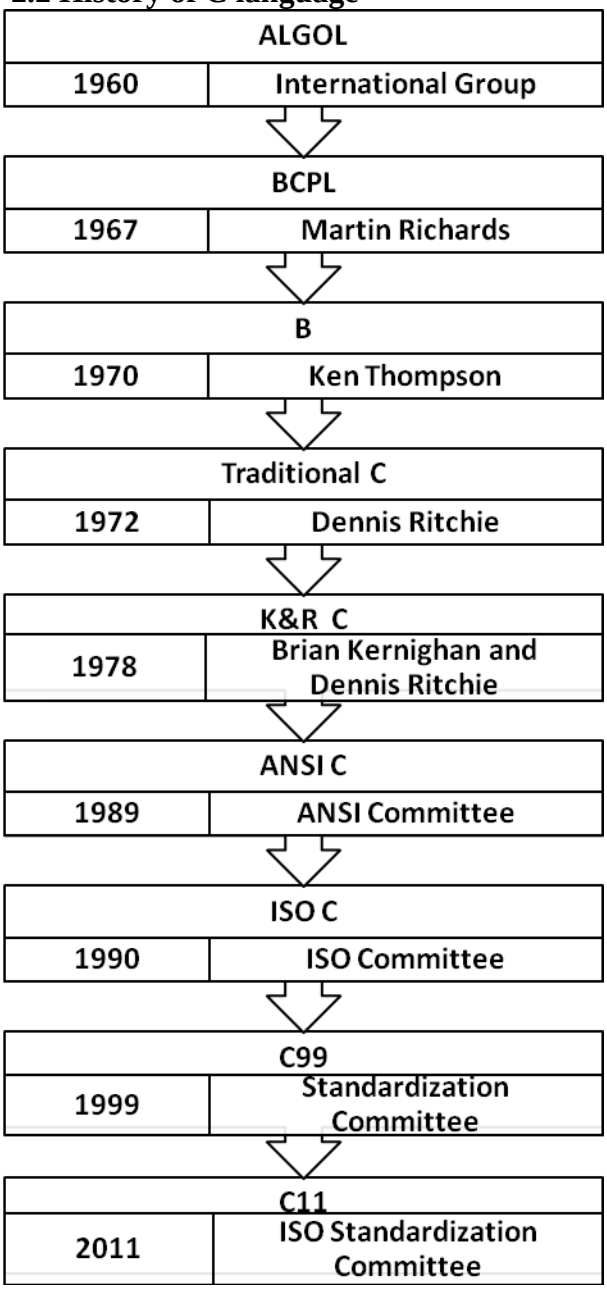



*Fig 2.1 Evolution of C language*

In 1972 a programmer, Dennis Ritchie, created a new language called C. C was not entirely a new language. It was derived from another programming language called B which in turn was derived from an old programming language called BCPL. C was designed with one goal in mind: writing operating systems. During the years 1972-73, Dennis Ritchie along with his colleagues at AT&T Bell Labs rewrote the entire UNIX operating system in C language which initially was written in assembly language.

Traditional C borrowed many concepts from ALGOL, BCPL and B. It added the concept of data types and other powerful features. It was a non-standardized version developed for internal usage inside AT&T and Bell Labs to mainly develop UNIX OS.

After the publication the of book "The C Programming Language" by Brian Kerninghan and Dennis Ritchie in 1978, the C language became popularly known as K&R C based on initials of its authors.

By late 1980's C became highly dominant in software industry. However each software vendor started developing and maintaining his own version of C and its associated libraries. This led to inconsistency and incompatibility issues. In order to tackle these issues, ANSI

(American National Standard Institute) released a standardized version of C known as ANSI C in 1989. ISO (International Organization for Standardization) approved ANSI C in 1990, and it became ISO C.

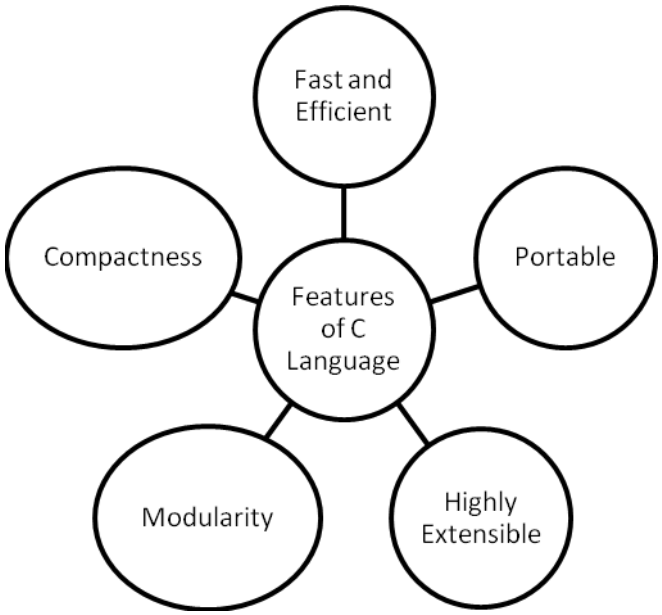
In 1999, ISO standardization committee updated C language according to the changing trends in the software industry and named it C99. C99 provides several new features, such as inline functions, new data types- long and long int, variable length arrays and support for single line comment.

The current standard of C is C11 which formalized by ISO standardization committee in 2011. C11 provides several enhancements over C99 including multithreading support, safer standard libraries, and better compliance with other industry standards

*Q 3. Write a short note on history and development of C?*

### 2.2.1 Features of C language

- C is very fast and efficient as it directly interacts with hardware and because it provides variety of operators and data types.
- C is portable as a program written in C can be run on a variety of devices




- C is highly extensible with the help of user defined libraries
- C is modular as its programs are always broken down and written with the help of functions
- C is a very compact language. Its original specification contained just 32 keywords and a very limited set of libraries. If a user wants more functionality he can always use external libraries.

*Q 4. Explain the features of C language?*

*Fig 2.2 Features of C language*

### 2.3 Advantages of C language

- Simple yet powerful
- High Performance
- Highly extensible
- Portable

### 2.4 Disadvantages of C language

- C is less sophisticated than newer languages like Java, C# or Python so the programmer need to write more lines of code to achieve similar results
- C does not support object oriented programming

- C does not support exception(run time error) handling
- C does not have strict type checking- e.g., we can pass an integer value for the float data type
- C does not have the concept of namespaces
- C does not have the concept of constructors and destructors

***Note**: The disadvantages of C will be clearer after you have thoroughly studied all the five units.*

*Q 5. Explain the advantages and disadvantages C language?*

### 2.5 Writing the First C Program

Let us consider a simple C program. Our first example displays a line of text. The program and its screen output are shown below:

```
1  //A simple program to print 'Hello World'
2  #include<stdio.h>
3
4  //main function starts execution
5  void main()
6  {
7        printf("Hello world\n");
8  }
9  //main function completes execution
```

**Output of the program:**

Hello World

Even though this program is simple, it illustrates several important features of the C language.

**2.5.1 Comments**: Lines 1, 4 and 9 begin with **//** (double forward slash), indicating that these two lines are comments. We insert comments to document programs and improve program readability. Comments do not cause the computer to perform any action when the program is run. Comments are ignored by the C compiler and do not cause any machine-language object code to be generated. The preceding comments simply describe the purpose of the program, from where the program starts execution and where the program stops execution. Comments also help other people read and understand our program.

**Note:** We can also use **/*…*/** multi-line comments in which everything from **/*** on the first line to ***/** at the end of the last line is a comment.

**2.5.2 Preprocessor Directives:** Line 2 (*#include<stdio.h>*) is a directive (command) to the C preprocessor which is a special program. Lines beginning with # are processed by the preprocessor before compilation. Line 2 tells the preprocessor to include the contents of the standard input/output header *(<stdio.h>*) in the program. This header contains information used by the compiler when compiling calls to standard input/output library functions such as printf(). A much detailed discussion about the c preprocessor will follow in unit 5.

**2.5.3 Blank Lines and White Space:** Line 3 is simply a blank line. You use blank lines, space characters and tab characters (i.e.,"tabs") to make programs easier to read. Together, these characters are known as whitespace. White-space characters are normally ignored by the compiler.

**2.5.4 The main Function:** Line 5 (void main() ) is called the main function. It is the point in a C program from where the execution starts. The parentheses after main indicate that main() is a program building block called a function. C programs contain one or more functions, one of which must always be main(). Every program in C begins executing at the function main. Functions can return information. The keyword void to the left of main indicates that here main() does not return any value.

A left brace, {, begins the body of every function (line 6). A corresponding right brace ends each function (line 8). This pair of braces and the portion of the program between the braces is called a **block**. A block is always executed in entirety i.e., either all the statements inside are executed or none of them is executed.

**2.5.5 An Output Statement:** Line 7 (printf) instructs the computer to perform an action, namely to print on the screen the string of characters enclosed within quotation marks. A string is sometimes called a character string,a message or a literal. The entire line, including the printf function (the "f" stands for"formatted"), its argument within the parentheses and the semicolon (;), is called a statement. Every statement must end with a semicolon (also known as the statement terminator).When the preceding printf statement is executed, it prints the message Welcome to C! on the screen. The characters normally print exactly as they appear between the double quotes in the printf statement. \n is a special symbol called the escape sequence that tells the compiler to move to the cursor to the next line after displaying the message on the screen.

## 2.6 Executing the first C program (Turbo C)

**Step 1**: Locate the TC.exe file and open it. You will usually find it at location **C:\TC\BIN\.**

**Step 2:** Goto File >New and then write your C program

**Step 3:** Save the program using $F_2$ (or File > Save), remember the extension should be either ".cpp" or ".c".

**Step 4:** Compile the program using Alt + F9 or Compile > Compile

**Step 5:** Press Ctrl + F9 to Run (or select Run > Run in menu bar ) the C program.

**Step 6:** Alt+F5 to view the output of the program at the output screen.

*Q 6. Write a short note on writing and executing the first C program?*

## 2.7 General Structure of a C Program

```
Documentation Section
Link Section
Definition Section
Global Declaration Section
void main()
{
        Declaration Section
        Executable part
}
Subprogram section
Function 1
Function 2
.
.
```

Function n

**The Documentation Section** consists of a set of comment lines giving the name of the program and other details.
**The Link Section** provides instructions to the compiler to link functions from the system library.
**The Definition Section** defines all symbolic constants.
**The Global Declaration Section:** There are some variables and those variables are declared in this section that is outside of all functions.
**main() function:** Every C program must have one main function section. This section contains two parts, declaration and executable part.

- **Declaration Part** declares all the variables used in the executable part.
- There should be at least one statement in the **executable part** which contains instructions to perform certain task.

The declaration and executable part must appear between the opening and closing braces. All statements in the declaration part should end with the semicolon.
**The Subprogram Section** contains all the user defined functions that are called in the main function.

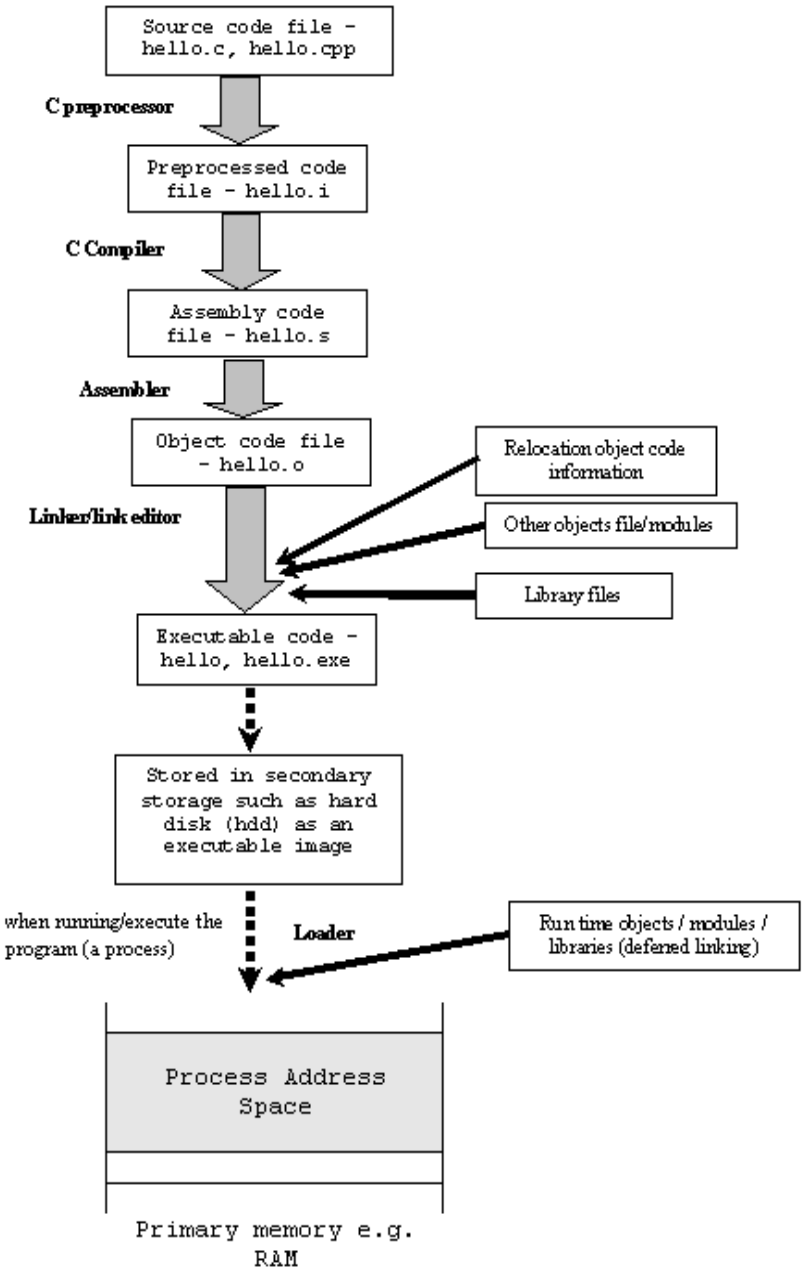
*Q 7. Describe the structure of a C program?*

## 2.8 Compilation of a C Program

The compilation process of a C program involves following steps:

1. **Preprocessing** is the first step of any C compilation. It processes include-files, conditional compilation instructions and macros.
2. **Compilation** is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.
3. **Assembly** is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.
4. **Linking** is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

There is one more step involved before actual execution of a C program takes place. It is called **loading**. In this step, the loader brings the C program from secondary storage (e.g., hard disk) to main memory (i.e., RAM) and prepares it for execution. It allocates memory and other necessary resources to the program and hands over the CPU control to it.




*Q 8. Explain the compilation process of a C program?**or***
*Q 9. Explain various components of C language?*

### 2.9 Standard Input/Output (I/O) in C

When we are saying **Input** that means to feed some data into program. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

When we are saying **Output** that means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output the data on the computer screen as well as you can save that data in text or binary files.

### 2.9.1 The Standard Files

C programming language treats all the devices as files. So devices such as the display are addressed in the same way as files and following three file are automatically opened when a program executes to provide access to the keyboard and screen.

| Standard File | File Pointer | Device |
|---|---|---|
| Standard input | stdin | Keyboard |
| Standard output | stdout | Screen |
| Standard error | stderr | Your screen |

The file points are the means to access the file for reading and writing purpose. This section will explain how to read values from the screen and how to print the result on the screen.

### 2.9.2 The getchar() & putchar() functions [Unformatted I/O]

The **int getchar(void)** function reads the next available character from the input. This function reads only single character at a time. We can use this method in the loop in case we want to read more than one characters from the screen.

The **int putchar(int c)** function puts the character on the screen. This function puts only single character at a time. We can use this method in the loop in case we want to display more than one character on the screen. Check the following example:

```
#include <stdio.h>
void main( )
{
   int c;
   printf( "Enter a value :");
   c = getchar( );
   printf( "\nYou entered: ");
   putchar( c );
}
```

When the above code is compiled and executed, it waits for you to input some text when you enter a text and press enter then program proceeds and reads only a single character and displays it as follows:

```
Enter a value : this is test
You entered: t
```

### 2.9.3 The gets() & puts() functions [Unformatted I/O]

The char *gets(char *s) function reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF (End of File).

The int puts(const char *s) function writes the string s and a trailing newline to stdout.

```
#include <stdio.h>
void main( )
{
   char str[100];
   printf( "Enter a value :");
   gets( str );
   printf( "\nYou entered: ");
```

```
  puts( str );
}
```

When the above code is compiled and executed, it waits for us to input some text when we enter a text and press enter then program proceeds and reads the complete line till end and displays it as follows:
Enter a value : this is test
You entered: This is test

### 2.9.4 The scanf() and printf() functions [Formatted I/O]

The **int scanf(const char *format, ...)** function reads input from the standard input stream stdin and scans that input according to format provided.
The **int printf(const char *format, ...)** function writes output to the standard output stream stdout and produces output according to a format provided.
The format can be a simple constant string, but we can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. For now let us proceed with a simple example which makes things clear:

```
#include <stdio.h>
void  main( )
{
  char str[100];
  int i;
  printf( "Enter a value :");
  scanf("%s %d", str, &i);
  printf( "\nYou entered: %s %d ", str, i);
}
```

When the above code is compiled and executed, it waits for us to input some text, when we enter a text and press enter then program proceeds and reads the input and displays it as follows:
Enter a value : seven 7
You entered: seven 7

Here, it should be noted that scanf() expect input in the same format as we provided %s and %d, which means we have to provide valid input like "string integer", if we provide "string string" or "integer integer" then it will be assumed as wrong input. Second, while reading a string scanf() stops reading as soon as it encounters a space so "this is test" are three strings for scanf().

#### 2.9.4.1 Format specifiers

Some important format specifiers in C:

- **%c** single character
- **%d** (%i) signed integer
- **%ld** long integer
- **%e** **(%E)** exponential format
- **%f** floating point decimal

**%lf** long float – long floating point decimal
**%o** unsigned octal value
**%p** pointer address stored in pointer
**%s** array of char sequence of characters
**%u** unsigned integer
**%x** **(%X)**unsigned hexadecimal value

**2.9.4.2 I/O of integers using printf()**
Let us consider following C program:

```
#include<stdio.h>
void main()
{
   int c=5;
   printf("Number=%d",c);
}
```

**Output**
Number=5
Inside quotation of printf() there, is a format specifier "%d" (for integer). If this format specifier matches with remaining argument,i.e, c in this case, value of c is displayed.
Let us consider one more example:

```
#include<stdio.h>
void main()
{
   int c;
   printf("Enter a number\n");
   scanf("%d",&c);
   printf("Number=%d",c);
}
```

**Output**

```
Enter a number
4
Number=4
```

The scanf() function is used to take input from user. In this program, the user is asked a input and value is stored in variable c. Note the '&' sign before c. &c denotes the address of c and value is stored in that address.

**2.9.4.3 I/O of floats in C**
Consider the following C program:

```
#include <stdio.h>
void main()
{
   float a;
   printf("Enter value: ");
```

```
   scanf("%f",&a);
   printf("Value=%f",a);    //%f is used for floats instead of %d
}
```

**Output**

```
Enter value: 23.45
Value=23.450000
```

Format specifier "%f" is used for floats to take input and to display floating value of a variable.

**2.9.4.4 I/O of Characters**

```
#include <stdio.h>
void main()
{
   char var1;
   printf("Enter character: ");
   scanf("%c",&var1);
   printf("You entered %c.",var1);
}
```

**Output**

```
Enter character: g
You entered g.
```

format specifier "%c" is used in case of characters.

**2.9.4.5 I/O of ASCII values**

When character is typed in the above program, the character itself is not recorded a numeric value(ASCII value) is stored. And when we displayed that value by using "%c", that character is displayed.

```
#include <stdio.h>
void main()
{
   char var1;
   printf("Enter character: ");
   scanf("%c",&var1);
   printf("You entered %c.\n",var1);
/* \n prints the next line(performs work of enter). */
   printf("ASCII value of %d",var1);
}
```

**Output**

```
Enter character:
```

g
103

When, 'g' is entered, ASCII value 103 is stored instead of g.

We can display character if we know ASCII code only. This is shown by following example.

```
#include <stdio.h>
void  main()
{
   int var1=69;
   printf("Character of ASCII value 69: %c",var1);
}
```

**Output**
Character of ASCII value 69: E
The ASCII value of 'A' is 65, 'B' is 66 and so on to 'Z' is 90. Similarly ASCII value of 'a' is 97, 'b' is 98 and so on to 'z' is 122.

**2.10.4.6 Variations in Output for integer and floats**
Integer and floating-points can be displayed in different formats in C programming as:

```
#include<stdio.h>
void  main()
{
   printf("Case 1:%6d\n",9876);
/*  Prints the number right justified within 6 columns  */
   printf("Case 2:%3d\n",9876);
/* Prints the number to be right justified to 3 columns but, there are 4 digits so number is not right justified  */
   printf("Case 3:%.2f\n",987.6543);
/* Prints the number rounded to two decimal places */
   printf("Case 4:%.f\n",987.6543);
/* Prints the number rounded to 0 decimal place, i.e, rounded to integer */
   printf("Case 5:%e\n",987.6543);
/* Prints the number in exponential notation(scientific notation) */
}
```

**Output:**

Case 1:  9876
Case 2:9876
Case 3:987.65
Case 4:988
Case 5:9.876543e+002

**2.9.4.7 Variations in Input for integer and floats**

```
#include <stdio.h>
void  main()
{
   int a,b;
   float c,d;
   printf("Enter two intgers: ");
/*Two integers can be taken from user at once as below*/
   scanf("%d%d",&a,&b);
   printf("Enter intger and floating point numbers: ");
/*Integer and floating point number can be taken at once from user as below*/
   scanf("%d%f",&a,&c);
}
```

Similarly, any number of input can be taken at once from user.

*Q 10. Write a short note on standard input/output (I/O) in C*

## 2.10 Constants, Variables and Datatypes

A program consists of set of instructions, the instructions are informed using certain symbols and words according to some rigid rules (or Grammar). Every program instructions must contain confirm precisely to the syntax, rules of the language. Like any other language C also has its own vocabulary and grammar.

### 2.10.1 Character set:

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

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

### 2.10.2 Constants:

Constants are identifiers whose value does not change. Constants are used to define fixed values like PI or the charge on an electron so that their value does not get changed in the program even by mistake. To declare a constant, precede the normal variable declaration with **const** keyword and assign it a value. For example,

```
const float pi = 3.14;
```

Another way to designate a constant is to use the pre-processor command ***define***.

```
#define PI 3.14159
```

When the preprocessor reformats the program to be compiled by the compiler, it replaces each defined name with its corresponding value wherever it is found in the source program. Hence, it just works like the Find and Replace command available in a text editor.

Constants refer to fixed value that does not change during the execution of a program. C constants can be divided into two major categories:

- Numeric constant (Integer constant and Real constant)
- Character constant (Single character constant and String constant)

Let us consider a problem involving usage of constants:

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

{
const int x=10;
printf("%d",x);
}
```

**2.10.3 Variables:**

An entity that may vary during program execution is called a variable. Variable names are names given to locations in memory. These locations can contain integer, real or character constants.

**Rules for Constructing Variable Names:**

- A variable name is any combination of 1 to 31 alphabets, digits or underscores.
- The first character in the variable name must be an alphabet or underscore.
- No commas or blanks are allowed within a variable name.
- No special symbol other than an underscore (as in **gross_sal**) can be used in a variable name.

e.g. simple_intrest,SUM, m1 etc.

**2.10.4 Keywords:**

Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). The keywords **cannot** be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. There are only 32 keywords available in C.

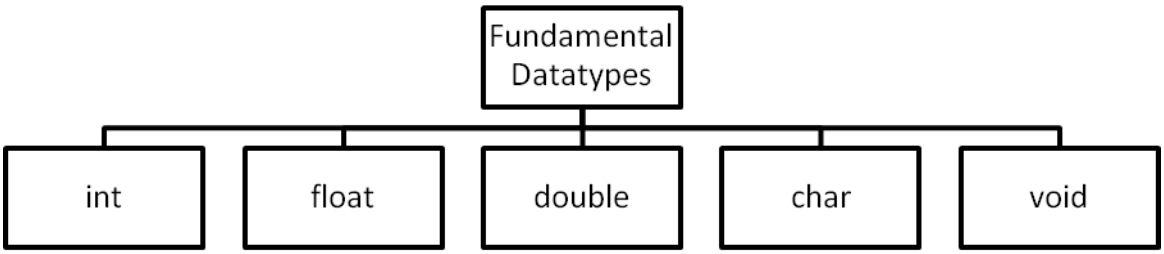
| | | | |
|---|---|---|---|
| auto | double | int | struct |
| break | else | long | switch |
| case | enum | register | typedef |
| char | extern | return | union |
| const | float | short | unsigned |
| continue | for | signed | void |
| default | goto | sizeof | volatile |
| do | if | static | while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

**2.10.5 Identifiers:**

Identifiers refer to the name of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as first character. Both upper case and lowercase letters are permitted. Underscore is permitted in identifier. Rules for naming an identifier are same as rules for naming a variable.

### 2.10.6 Fundamental Datatypes in C

They are also called Primitive Datatypes.




#### 2.10.6.1 Integer Datatype

**Keyword:** int
**Minimum Range:** -32768 to 32767
**Format specifier:** %d

An integer is a whole number (a number without a fractional part). It can be positive or negative numbers like 1, -2, 3, etc., or 0.The sizes of the integer variables depend on the hardware and operating system of the computer.

#### 2.10.6.2 Floating Point Datatype

**Keyword:** float
**Minimum Range:**-3.4e38 to +3.4e38
**Format specifier:** %f

Floating point numbers are numbers with a decimal point. The float type can take large floating point numbers with a small degree of precision (Precision is simply the number of decimal places to which a number can be calculated with accuracy. If a number can be calculated to three decimal places, it is said to have three significant digits.)

#### 2.10.6.3 Double Floating Datatype

**Keyword:** double
**Minimum Range:**-1.7e308 to +1.7e308
**Format specifier:** %lf

Double-precision floating point numbers are also numbers with a decimal point. We know that the float type can take large floating point numbers with a small degree of precision but the double-precision double type can hold even larger numbers with a higher degree of precision.

#### 2.10.6.4 Character Datatype
**Keyword:**char

**Minimum Range:**-128 to +127
**Format specifier:** %c
char is a special integer type designed for storing single characters. The integer value of a char corresponds to an ASCII (*American Standard Code for Information Interchange*) character. E.g., a value of 65 corresponds to the letter A, 66 corresponds to B, 67 to C, and so on.

**2.10.6. 5 Void Datatype**
**Keyword:** void
**Minimum Range:** nil
**Format specifier:** nil

The type specifier void indicates that no value is available. It has three main functions
**1. Function returns as void**
There are various functions in C which do not return value or we can say they return void. A function with no return value has the return type as void. For example void exit (int status);
**2. Function arguments as void**
There are various functions in C which do not accept any parameter. A function with no parameter can accept as a void. For example, int rand(void);
**3. Pointers to void**
A pointer of type void * represents the address of an object, but not its type. For example a memory allocation function void *malloc( size_t size ); returns a pointer to void which can be casted to any data type.

**2.10.7 Extended Datatypes or Type Modifiers**
The fundamental data types explained above have the following modifiers.

- short
- long
- signed
- unsigned

The modifiers define the amount of storage allocated to the variable. The amount of storage allocated is not cast in stone. ANSI has the following rules:

short int <=int <= long int
float <= double <= long double

What this means is that a 'short int' should assign less than or the same amount of storage as an 'int' and the 'int' should be less or the same bytes than a 'long int'. What this means in the real world is:

| **Type** | **Bytes** | **Range** | |
|---|---|---|---|
| short int | 2 | -32,768 -> +32,767 | (32kb) |
| unsigned short int | 2 | 0 -> +65,535 | (64Kb) |
| unsigned int | 4 | 0 -> +4,294,967,295 | (4Gb) |
| int | 4 | -2,147,483,648 -> +2,147,483,647 | (2Gb) |
| long int | 4 | -2,147,483,648 -> +2,147,483,647 | (2Gb) |
| signed char | 1 | -128 -> +127 | |
| unsigned char | 1 | 0 -> +255 | |
| float | 4 | | |
| double | 8 | | |
| long double | 12 | | |

These figures only apply to today's generation of PCs. Mainframes and midrange machines could use different figures, but would still comply with the rule above. We can find out how much storage is allocated to a data type by using the **sizeof** operator.
Here is an example to check size of memory taken by various datatypes.

```
#include<stdio.h>
void main()
{
  printf("sizeof(char) == %d\n", sizeof(char));
  printf("sizeof(short) == %d\n", sizeof(short));
  printf("sizeof(int) == %d\n", sizeof(int));
  printf("sizeof(long) == %d\n", sizeof(long));
  printf("sizeof(float) == %d\n", sizeof(float));
  printf("sizeof(double) == %d\n", sizeof(double));
  printf("sizeof(long double) == %d\n", sizeof(long double));
  printf("sizeof(long long) == %d\n", sizeof(long long));
}
```

### 2.10.8 Type Qualifiers

A type qualifier is used to refine the declaration of a variable, a function, and parameters, by specifying whether:

- The value of a variable can be changed.
- The value of a variable must always be read from memory rather than from a register

Standard C language recognizes the following two qualifiers:

- const
- volatile

The const qualifier is used to tell C that the variable value cannot change after initialization.

```
const float pi=3.14159;
```

Now pi cannot be changed at a later time within the program. Another way to define constants is with the #define preprocessor which has the advantage that it does not use any storage

The volatile qualifier declares a data type that can have its value changed in ways outside the control or detection of the compiler (such as a variable updated by the system clock or by another program). This prevents the compiler from optimizing code referring to the object by storing the object's value in a register and re-reading it from there, rather than from memory, where it may have changed. You will use this qualifier once you will become expert in "C". So for now just proceed.

*Q 11. What are variables, constants, identifiers and keywords?*
*Q 12. Enumerate rules for naming variables and identifiers?*
*Q 13. Write a short note on fundamental and extended data types in C language?*

Every variable in C programming has two properties: type and storage class. Type refers to the data type of variable whether it is character or integer or floating-point value etc. And storage class determines how long it stays in existence.

- The storage class of a variable defines the scope (visibility) and life time of variables and/or functions declared within a C Program. In addition to this, the storage class gives the following information about the variable or the function.
- It is used to determine the part of memory where storage space will be allocated for that variable or function (whether the variable/function will be stored in a register or in RAM)
- it specifies how long the storage allocation will continue to exist for that function or variable.
- It specifies the scope of the variable or function. That is, the part of the C program in which the variable name is visible, or accessible.
- It specifies whether the variable or function has internal, external, or no linkage
- It specifies whether the variable will be automatically initialized to zero or to any indeterminate value

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

### 2.11.1 Automatic storage class

Keyword for automatic variable

*auto*

Variables declared inside the function body are automatic by default. These variable are also known as local variables as they are local to the function and doesn't have meaning outside that function

Since, variable inside a function is automatic by default, keyword auto are rarely used.

### 2.11.2 External storage class

External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.

Example to demonstrate working of external variable

```
void Check();
int a=5;
/* a is global variable because it is outside every function */
int main(){
    a+=4;
    Check();
    return 0;
}
void Check(){
   ++a;
/*  ----- Variable a is not declared in this function but, works in any function as they are global
variable -------  */
   printf("a=%d\n",a);
}
```

**Output**

```
a=10
```

### 2.11.3 Register Storage Class

Keyword to declare register variable

```
register
```

Example of register variable

```
register int a;
```

Register variables are similar to automatic variable and exists inside that particular function only.
If the compiler encounters register variable, it tries to store variable in microprocessor's register rather than memory. Value stored in register are much faster than that of memory.
In case of larger program, variables that are used in loops and function parameters are declared register variables.
Since, there are limited number of register in processor and if it couldn't store the variable in register, it will automatically store it in memory.

### 2.11.4 Static Storage Class

The value of static variable persists until the end of the program. A variable can be declared static using keyword: *static*. For example:

```
static int i;
```

Here, *i* is a static variable.

```
#include <stdio.h>
void Check();
int main(){
   Check();
   Check();
   Check();
}
void Check(){
    static int c=0;
    printf("%d\t",c);
    c=c+5;
}
```

**Output**

```
0      5     10
```

During first function call, it will display 0. Then, during second function call, variable *c* will not be initialized to 0 again, as it is static variable. So, 5 is displayed in second function call and 10 in third call.
If variable c had been automatic variable, the output would have been:

```
0     0     0
```

| FEATURE | STORAGE CLASS | | | |
|---|---|---|---|---|
| | **Auto** | **Extern** | **Register** | **Static** |
| **Accessibility** | Accessible within the function or block in which it is declared | Accessible within all program files that are a part of the program | Accessible within the function or block in which it is declared | Local: Accessible within the function or block in which it is declared<br>Global: Accessible within the program in which it is declared |
| **Storage** | Main Memory | Main Memory | CPU Register | Main Memory |
| **Existence** | Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared | Exists throughout the execution of the program | Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared | Local: Retains value between function calls or block entries<br>Global: Preserves value in program files |
| **Default value** | Garbage | Zero | Garbage | Zero |

*Q 14. Explain various storage classes in C*

## 2.12 Operators in C

C language supports a lot of operators to be used in expressions. These operators can be categorized into the following major groups:

- Arithmetic operators
- Relational Operators
- Equality Operators
- Logical Operators
- Unary Operators
- Conditional Operators
- Bitwise Operators
- Assignment operators
- Comma Operator
- Sizeof Operator

### 2.13.1 Arithmetic Operators

| Operation | Operator | Syntax | Comment | Result |
|---|---|---|---|---|
| Multiply | * | a * b | result = a * b | 27 |
| Divide | / | a / b | result = a / b | 3 |
| Addition | + | a + b | result = a + b | 12 |
| Subtraction | - | a - b | result = a – b | 6 |
| Modulus | % | a % b | result = a % b | 0 |

### 2.13.2 Relational Operators

Also known as a comparison operator, it is an operator that compares two values. Expressions that contain relational operators are called *relational expressions.* Relational operators return true or false value, depending on whether the conditional relationship between the two operands holds or not.

| Operator | Meaning | Example |
|---|---|---|
| < | Less Than | 3 < 5 Gives 1 |
| > | Greater Than | 7 > 9 Gives 0 |
| >= | Less Than Or Equal To | 100 >= 100 Gives 1 |
| <= | Greater Than Equal To | 50 >=100 Gives 0 |

### 2.13.3 Equality Operators

C language supports two kinds of equality operators to compare their operands for strict equality or inequality. They are equal to (==) and not equal to (!=) operator. The equality operators have lower precedence than the relational operators.

| Operator | Meaning |
|---|---|
| == | RETURNS 1 IF BOTH OPERANDS ARE EQUAL, 0 OTHERWISE |
| != | RETURNS 1 IF OPERANDS DO NOT HAVE THE SAME VALUE, 0 OTHERWISE |

### 2.13.4 Logical Operators

C language supports three logical operators. They are- Logical AND (&&), Logical OR (||) and Logical NOT (!).As in case of arithmetic expressions, the logical expressions are evaluated from left to right.

| A | B | A&&B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| A | B | A\|\|B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| A | !A |
|---|---|
| 0 | 1 |
| 1 | 0 |

### 2.13.5 Unary Operators

Unary operators act on single operands. C language supports three unary operators. They are unary minus, increment and decrement operators. When an operand is preceded by a minus sign, the unary operator negates its value. The increment operator is a unary operator that increases the value of its operand by 1. Similarly, the decrement operator decreases the value of its operand by 1. For example,

```
int x = 10, y;
y = x++;
```

is equivalent to writing

```
y = x;
x = x + 1;
```

whereas, y = ++x;
is equivalent to writing
x = x + 1;
y = x;

### 2.13.6 Conditional Operator (Ternary Operator)

The conditional operator(?:) is just like an if .. else statement that can be written within expressions. The syntax of the conditional operator is

exp1 ? exp2 : exp3

Here, exp1 is evaluated first. If it is true then exp2 is evaluated and becomes the result of the expression, otherwise exp3 is evaluated and becomes the result of the expression. For example,

large = ( a > b) ? a : b

Conditional operators make the program code more compact, more readable, and safer to use as it is easier both to check and guarantee that the arguments that are used for evaluation.
Conditional operator is also known as ternary operator as it is neither a unary nor a binary operator; it takes *three* operands.

### 2.13.7 Bitwise Operators

Bitwise operators perform operations at bit level. These operators include: bitwise AND, bitwise OR, bitwise XOR and shift operators.

- The bitwise AND operator (&) is a small version of the boolean AND (&&) as it performs operation on bits instead of bytes, chars, integers, etc.

| x | y | x & y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |

| | | |
|---|---|---|
| 1 | 1 | 1 |

- The bitwise OR operator (|) is a small version of the boolean OR (||) as it performs operation on bits instead of bytes, chars, integers, etc.

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

- The bitwise NOT (~), or complement, is a unary operation that performs logical negation on each bit of the operand. By performing negation of each bit, it actually produces the ones' complement of the given binary value.

| x | ~ x |
|---|---|
| 0 | 1 |
| 1 | 0 |

- The bitwise XOR operator (^) performs operation on individual bits of the operands. The result of XOR operation is shown in the table

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

**2.13.8 Bitwise Shift Operators**

In bitwise shift operations, the digits are moved, or *shifted*, to the left or right. The CPU registers have a fixed number of available bits for storing numerals, so when we perform shift operations; some bits will be "shifted out" of the register at one end, while the same number of bits are "shifted in" from the other end.

In a left arithmetic shift, zeros are shifted in on the right. For example;

unsigned int x = 11000101;

Then x << 2 = 00010100

If a right arithmetic shift is performed on an unsigned integer then zeros are shifted on the left.

unsigned int x = 11000101;

Then x >> 2 = 00110001

**2.13.9 Assignment Operators**

The assignment operator is responsible for assigning values to the variables. While the equal sign (=) is the fundamental assignment operator, C also supports other assignment operators that provide shorthand ways to represent common variable assignments. They are shown in the table.

| Operator | Syntax | Equivalent To |
|---|---|---|
| /= | variable /= expression | variable = variable / expression |
| \= | variable \= expression | variable = variable \ expression |

| | | |
|---|---|---|
| *= | variable *= expression | variable = variable * expression |
| += | variable += expression | variable = variable + expression |
| -= | variable -= expression | variable = variable – expression |
| &= | variable &= expression | variable = variable & expression |
| ^= | variable ^= expression | variable = variable ^ expression |
| <<= | variable <<= amount | variable = variable << amount |
| >>= | variable >>= amount | variable = variable >> amount |

### 2.13.10 Comma Operator

The comma operator in C takes two operands. It works by evaluating the first and discarding its value, and then evaluates the second and returns the value as the result of the expression. Comma separated operands when chained together are evaluated in left-to-right sequence with the right-most value yielding the result of the expression. Among all the operators, the comma operator has the lowest precedence. For example,

```
int a=2, b=3, x=0;
x = (++a, b+=a);
```

Now, the value of x = 6.

### 2.13.11 Sizeof Operator

sizeof is a unary operator used to calculate the sizes of data types. It can be applied to all data types. The operator returns the size of the variable, data type or expression in bytes. 'sizeof' operator is used to determine the amount of memory space that the variable/expression/data type will take. For example,

sizeof(char) returns 1, that is the size of a character data type. If we have,

```
int a = 10;
unsigned int result;
result = sizeof(a);
```

then result = 2

*Q 15. Write short note on various category of operators present in C*

## 2.13 Type Casting (or Type Conversion)

Type casting is a way to convert a variable from one data type to another data type. For example, if we want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another explicitly using the **cast operator** as follows:

*(type_name) expression*

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation:

```
#include <stdio.h>
void main()
{
  int sum = 17, count = 5;
  double mean;
  mean = (double) sum / count;
```

```
   printf("Value of mean : %f\n", mean );

}
```
When the above code is compiled and executed, it produces the following result:
Value of mean : 3.400000

It should be noted here that the cast operator has precedence over division, so the value of **sum** is first converted to type **double** and finally it gets divided by count yielding a double value.
Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the **cast operator**. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

### 2.14.1 Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than **int** or **unsigned int** are converted either to **int** or **unsigned int**. Consider an example of adding a character in an int:

```
#include <stdio.h>
void main()
{
   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   int sum;
   sum = i + c;
   printf("Value of sum : %d\n", sum );
}
```
When the above code is compiled and executed, it produces the following result:
Value of sum : 116
Here, value of sum is coming as 116 because compiler is doing integer promotion and converting the value of 'c' to ascii before performing actual addition operation.

### 2.14.2 Usual Arithmetic Conversion

The **usual arithmetic conversions** are implicitly performed to cast their values in a common type. Compiler first performs *integer promotion*, if operands still have different types then they are converted to the type that appears highest in the following hierarchy:

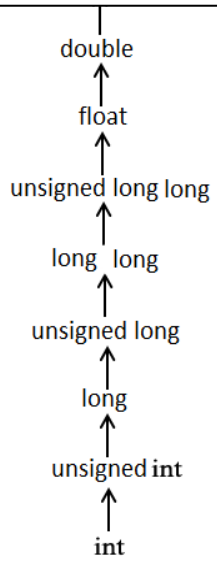
The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take following example to understand the concept:

```
#include <stdio.h>
void main()
{
   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   float sum;
   sum = i + c;
   printf("Value of sum : %f\n", sum );
}
```

When the above code is compiled and executed, it produces the following result:
Value of sum : 116.000000

Here, it is simple to understand that first c gets converted to integer but because final value is double, so usual arithmetic conversion applies and compiler convert i and c into float and add them yielding a float result.

*Q 16. What is typecasting? Explain various types of typecasting mechanism present in C*




## 2.14 Precedence of Operators

It decides the order of operators during evaluation of an expression, C language has predefined rule of priority of operators. This rule of priority of operators is called operator precedence.
In C, precedence of arithmetic operators(*,%,/,+,-) is higher than relational operators(==,!=,>,<,>=,<=) and precedence of relational operator is higher than logical operators(&&, || and !). Suppose an expression:

(a>b+c&&d)

This expression is equivalent to:

((a>(b+c))&&d)

i.e, (b+c) executes first
then, (a>(b+c)) executes
then, (a>(b+c))&&d) executes

(a>b+c&&d)

This expression is equivalent to:

((a>(b+c))&&d)

i.e, (b+c) executes first
then, (a>(b+c)) executes
then, (a>(b+c))&&d) executes

## 2.15 Associativity of Operators

Associativity indicates in which order two operators of same precedence(priority) executes. Let us suppose an expression:

a==b!=c

Here, operators == and != have same precedence. The associativity of both == and != is left to right, i.e, the expression in left is executed first and execution take pale towards right. Thus, a==b!=cequivalent to :

(a==b)!=c

The table below shows all the operators in C with precedence and associativity.

**Note:** Precedence of operators decreases from top to bottom in the given table.

| **Summary of C operators with precedence and associativity** | | |
|---|---|---|
| **Operator** | **Meaning of operator** | **Associativity** |
| ()<br>[]<br>-><br>. | Functional call<br>Array element reference<br>Indirect member selection<br>Direct member selection | Left to right |
| !<br>~<br>+<br>-<br>++<br>--<br>&<br>*<br>sizeof<br>(type) | Logical negation<br>Bitwise(1 's) complement<br>Unary plus<br>Unary minus<br>Increment<br>Decrement<br>Dereference Operator(Address)<br>Pointer reference<br>Returns the size of an object<br>Type cast(conversion) | Right to left |
| *<br>/<br>% | Multiply<br>Divide<br>Remainder | Left to right |
| +<br>- | Binary plus(Addition)<br>Binary minus(subtraction) | Left to right |
| <<<br>>> | Left shift<br>Right shift | Left to right |
| <<br><=<br>><br>>= | Less than<br>Less than or equal<br>Greater than<br>Greater than or equal | Left to right |
| ==<br>!= | Equal to<br>Not equal to | Left to right |
| & | Bitwise AND | Left to right |
| ^ | Bitwise exclusive OR | Left to right |
| \| | Bitwise OR | Left to right |
| && | Logical AND | Left to right |
| \|\| | Logical OR | Left to right |
| ?: | Conditional Operator | Left to right |

| | | |
|---|---|---|
| =<br>*=<br>/=<br>%=<br>-=<br>&=<br>^=<br>\|=<br><<=<br>>>= | Simple assignment<br>Assign product<br>Assign quotient<br>Assign remainder<br>Assign sum<br>Assign difference<br>Assign bitwise AND<br>Assign bitwise XOR<br>Assign bitwise OR<br>Assign left shift<br>Assign right shift | Right to left |
| , | Separator of expressions | Left to right |

*Q 17. What is operator precedence and associativity? Explain with examples?*

**Questions**

1. In C programming what will be the value of r if r=p%q where p=-17 and q=5. [2011-12]
2. In C programming, what will be the output of the following code? Explain your answer. [2011-12]
3. What are identifiers, variables and constants? Mention rules to construct an identifier. Give some examples. [2012-13]
4. What is the meaning of scope of a variable? Give various types of scope in C programming? [2011-12]
5. Write a Program (WAP) in C in which values of variables x, y, x are input by the user, then their values are rotated such that x has value of y, y has value of z, and z has value of y. [2011-12]
6. Explain datatypes in 'C' language, mentioning their range, space they occupy in memory and keyword used for their representation in memory. [2011-12][2015-16]
7. Explain four storage classes in C, mentioning their place of storage, default initial value, scope and life of each item [2011-12][2013-14]]2014-15][2015-16]
8. What do you mean by operator precedence and associativity. [2013-14][2015-16]
9. What do you mean by implicit and explicit typecasting? [2013-14][2015-16]
10. What is difference between type conversion and type casting. What are the escape sequences characters? [2015-16]
11. Give any four format specifiers used in printf() [2013-14]
12. Explain the ternary operator in detail with example. [2012-13]
13. Explain various operators present in C?
14. Explain the general structure or layout of a C program? [2015-16]
15. Explain logical and bit operators with example. [2014-15]
16. Write about the formatted and unformatted Input/Output functions in C. [2014-15]
17. Explain the dot (.) operator in C language with proper example? [2015-16]
18. Write a short note on structured programming. [2015-16]
19. Explain loading and linking of a program in detail. [2015-16]