# *C-01 Programming with C - General Concepts*

## 1.1 Introduction

C is a programming language developed at AT&T's Bell Laboratories of USA in 1972. It was designed and written by Dennis Ritchie and it slowly replaced other contemporary languages like PL/I, Algol, Pascal, & APL. Since then C has become a popular programming language as it is reliable, simple and easy to use.

Earlier specific programming languages were developed for specific purposes like COBOL was developed for commercial applications, FORTRAN for scientific applications etc. This led to people learning different languages for different purposes. To avoid this, the need for a common language that could do all types of jobs was felt and ALGOL60 was developed to meet the purpose. The series of programming languages that led to the development of C after ALGOL60 include – CPL, BCPL, B and finally Ritchie developed C by combining the features of B & BCPL along with his own ideas.

There are two types of programming languages:

1. Problem Oriented Languages or **High Level Languages** like Fortran, Basic, Pascal etc. &
2. Machine Oriented Languages or **Low Level Languages** like Assembly Language, Machine Language etc.

C is a language that stands in-between these two types and is often called a **Middle Level Language** as it was designed to have the positive points of both the types of languages. To learn C we have to first stepwise learn what the alphabets, numbers & special symbols that are used in C. Then we will learn how constants, variables and keywords are constructed using the alphabets and finally how all these are combined together to form an instruction. When a single or several groups of instructions are combined together, we get a program.

## 1.2 Structure of a C Program

Every C program consists of one or more **modules** called functions, which are a series of instructions to the computer to perform a specific task. Many functions that we will be using are already written and compiled and are available in the function libraries supplied with the compiler. Thus instead of writing all individual instructions, one just tells the compiler to use one of its standard functions. Only when one wants to perform a task that is not in the function library, one has to write his function.

For example though C does not have any inbuilt command to display characters on the screen it has a function called **`printf()`** to do the same job. Hence, one does not have to always write a function each time one wants to display something on the screen but will have to only call the function **`printf()`** from the function library to do the job. Whereas suppose if someone wants to display some fixed message with the help of a function, he has to write his custom function and save it for future use. Whenever required, he will simply have to call the function created by him and it will display the required message on the screen.

Now let us come to the structure of a C program. In writing a C program, lowercase alphabets are used to denote any key/command word, function names and variables where capital letters are used to indicate constants.

As a C program consists of modules called functions, every C program must begin with the function called "**`void main()`**". The parenthesis "**`()`**"after the "**`main`**" is a part of the syntax for a function name and is necessary. The program begins by executing the **`void main()` function first**. Following the **`void main()`** function are the instructions, that can include C commands, function names in the library or functions written by the programmer. The **`void main()`** function executes the instructions one by one and accesses any other function as per the need of the program.

- **Delimiters**: An opening **brace** "**`{`**" must appear before the first instruction and a closing brace "**`}`**" must follow the last instruction. The opening and closing braces are called delimiters and they mark the beginning and end of a block. Every function one writes must start and end with braces. Again within a function there can be blocks of code that have their own braces.
- **Statement Terminator**: Each instruction in a C program must end with a **semicolon** "**`;`**" which is also known as a statement terminator. The semicolon tells the compiler that it has reached the end of the instruction and what comes next will be another instruction or the end of the program. However this does not imply that every line must end with a semicolon rather **every instruction should end with a semicolon** and an instruction can consist of more than one line on the screen.

So the basic structure of a C program looks like this:

```
void main()                ⇒ Indicates this to be the first function to start
{                          ⇒ The main function starts here with a delimiter
instruction-1 ;            ⇒ Statement terminator
instruction-2;
... ... ...                ⇒ These are the instructions to be performed
instruction-n;
}                          ⇒ The main function ends here with a delimiter
```

- **The `return` statement**: when in general a computer finishes performing the instructions, the program stops and the computer returns to the state it was in before the program. In general the return of control to the system takes place automatically. The **`return`** statement causes the program logic to return to the point from which the function was accessed. This return of control takes place automatically but for some compilers the **`return`** statement needs to be inserted just before the closing braces of the **`void main()`** function. The '0' after the return keyword indicates to the operating system that the program has terminated successfully. (The brackets for **(**0**)** are optional for some compilers).

- **The `#include` directive**: When using library functions in a program, certain specific information needs to be included within the main portion of the program. These information are generally stored in specific files which are supplied with the compiler and can be accessed using the **`#include`** command. The files that need to be included usually have a "**.h**" extension and generally designates a **header file**, indicating that the file needs to be included at the beginning of the program before the **`void main()`** function. As an example the command to include the header file named stdio.h will look like:

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

  This command is a **directive** that tells the compiler to use the information in the header file called **`stdio.h`** [the initials *stdio* stand for "**st**an**d**ard **i**nput **o**utput"] and the contents of the file need to be inserted into the program before the beginning of the compilation process. In this example the header file stdio.h contains all the information the compiler needs to work with disk files and send information to the screen and printer.

  Surrounding the name of the header file with the symbols **<**...**>** tells the compiler that the file may be located in the default "include directory". This is the folder where the compiler's installation program places the header files.

  Some built-in C functions also need the stdio.h file to run properly. For example the function **`getc()`** inputs a single character from a source one designates such as a file on the disk or the keyboard. There is another function called **`getch()`** which inputs a single character from the keyboard. It does so by calling **`getc()`** and telling it that the source is the standard input device. However since the standard input device is defined in the file stdio.h, to use the **`getch()`** function the **`#include<stdio.h>`** should be included, otherwise the compilation will show error. Thus the header files and the library functions work hand-in-hand to complete the function.

- **Comments** /* **…** */: For a better understanding of the program logic sometimes it may be helpful to add comments in a program. Accordingly the program name, programmer name, the purpose of the program and any other relevant information may need to be included in a program. Any such information which is not a part of the coding can be incorporated inside the program by writing those comments in between the symbols **`/*`** and **`*/`**. Whenever the compiler encounters the symbol combination /* and */ it ignores whatever is written within them. An example is given below:

  ```
  /*PROGRAM1: To calculate the average of a set of numbers*/
  ```

Now it is high time we write some preliminary C programs to practice what we have learnt so far. Apart from the functions we have learnt, we will be using three new functions to print data on the screen and to intake data from the keyboard. These are **`puts()`**, **`printf()`**, **`getch()`** and **`scanf()`**. The functions will be discussed in detail at a later stage.

- **Example1**: To print Name, School, Class & Section on the screen

```
/*Program-01*/             ⇒ Comment to indicate program name
#include <stdio.h>          ⇒ Include the standard i/p-o/p header file
void main()                 ⇒ The function name & start of the program
{                           ⇒ Opening brace indicating start of block of instruction
puts("College: George College");
puts("Name: Sukalyan Som");          Four instructions
puts("Class: BCA");          each one ending with a ";"
```

```
puts("Hello everybody!");
```
⇒ Return statement indicating end of program

```
}
```
⇒ Closing brace indicating end of block of instructions

In example-1 we have used the **puts()** function to display the data on the screen. Whatever string is there within the quotes " " gets displayed on the screen.

- **Example2**: To find average of three numbers

```
/*Program-02:To find Average of three numbers*/
#include <stdio.h>
void main()
{
printf("The average of the 3 numbers is:=%f",(5+19+3)/3);
```
⇒ Calculation

```
}
```

- **Example3**: To find the area of a circle

```
/*Program-03: To find Area of a Circle*/
#include <stdio.h>
void main()
{
float radius, area;
```
⇒ Declaration of variables with variable type as float

```
printf("Radius = ? :");
```
⇒ Input asked for radius

```
scanf("%f", &radius);
```
⇒ Reads the input and stores it in the variable radius

```
area=3.1416*radius*radius;
```
⇒ The area is calculated

```
printf("\nArea = %f", area);
```
⇒ The calculated area is printed on the screen

```
}
```

In the above examples 2 & 3, **float** type variables (numbers with a value after the decimal point like 2.54) are used along with **Format Place Holder** **%f** to input or output the required floating point numbers.

- **Example4**: To find the $n^{th}$ term of an arithmetic progression

```
/*Program-04: To find n-th term of an A.P.*/
#include <stdio.h>
void main()
{
int term_1, diff, term_n, n;
```
⇒ Declares 4 int type variables term_1, term_n, diff, & n

```
printf("*To find the n-th term of an A.P.*");
printf("\nFirst Term = ? :");
scanf("%d", &term_1);
```
⇒ **Note**: Put an "**&**" before the name of the input variable

```
printf("\nCommon Difference = ? :");
scanf("%d", &diff);
printf("\nTerm to find = ? :");
scanf("%d", &n);
term_n=term_1+(n-1)*diff;
printf("\nn-th term of the A.P. series = %d", term_n);
printf("Press ENTER to exit");
getch();
```
⇒ Waits for an input before leaving the screen

```
}
```

In the above examples the combination **\n** has occurred several times. It is known as an escape sequence and it starts a new line before typing whatever there is after it. Escape sequences will be discussed in detail at a later stage. Presently we will be using these to start a new line in our programs. The **getch()** function will also be discussed later. Presently it is used to halt the program until the user presses any key to continue further. This helps the user to view the output before the program execution ends.
Let us now find out the output of the program in the example-2 to get a better understanding of how C works.

| Program-3 | Output on Screen | Keyboard Input |
|---|---|---|
| `/*Program-03:To find Area of a Circle*/`<br>`#include <stdio.h>` | | |
| | | |

| | | |
|---|---|---|
| `void main()` | | |
| `{` | | |
| `float radius, area;` | | |
| `printf("Radius = ? : ");` | `Radius = ? :` | |
| `scanf("%f", &radius);` | `Radius = ? : 2.5` ← | 2.5 |
| `area=3.1416*radius*radius;` | | |
| `printf("\nArea = %f", area);` | `Area = 19.635000` | |
| | | |
| `}` | | |

Similarly the output of the program for example 4 will be:

| Program-4 | Output on Screen | Keyboard Input |
|---|---|---|
| `/*Program-04: To find n-th term of an A.P.*/` | | |
| `#include <stdio.h>` | | |
| `void main()` | | |
| `{` | | |
| `int term_1, diff, term_n, n;` | | |
| `printf("*To find the n-th term of an A.P.*");` | `*To find the n-th term of an A.P.*` | |
| `printf("First Term = ? : ");` | `First Term = ? :` | |
| `scanf("%d", &term_1);` | `First Term = ? : 2` ← | 2 |
| `printf("\nCommon Difference = ? : ");` | `Common Difference = ? :` | |
| `scanf("%d", &diff);` | `Common Difference = ? : 3` ← | 3 |
| `printf("\nTerm to find = ? : ");` | `Term to find = ? :` | |
| `scanf("%d", &n);` | `Term to find = ? : 5` ← | 5 |
| `term_n=term_1+(n-1)*diff;` | | |
| `printf("\nn-th term of the A.P.= %d", term_n);` | `n-th term of the A.P.= 14` | |
| `printf("Press ENTER to exit");` | `Press ENTER to exit` | |
| `getch();` | | |
| | | |
| `}` | | |

## 1.3 Types of Data

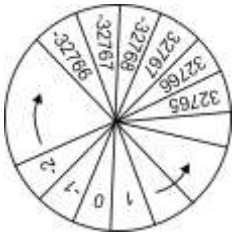
Information given to a computer is generally known as data. Data is given to a computer, which processes it and gives information as output. But before supplying data to a computer it must be told what type of data it is dealing with so that it can set aside sufficient space to store each data item without wasting any memory. (It can be noted here that in earlier days memory was really costly and one of the aims of a good program was to minimise the requirement of memory).

We specify a data item by the type of data it holds. In the examples 3 & 4 we have used the terms **`float`** & **`int`** to specify that the data that will be input from the keyboard will be of **`float`** type and **`int`** type. We now state below the different types of data that are possible in C.

- **Integer** type Data: represented as **int (2 bytes long):**

  An integer number is a number with **no decimal point** i.e. a whole number. It can be a positive or negative number or zero. It is used generally for counting purposes or for calculations, which involve only integer values or have results, which are integer in nature. Each piece of integer data requires two characters or **2 bytes of memory spaces** i.e. 16 bits. Hence the total number of values possible is equal to $(2^{16}-1) = 65536$ and the range of integers include:

  0 to 65535 for unsigned numbers or from –32768 to 32767 (32768+0+32767=65565) including 0 for signed numbers. For signed numbers think of the numbers as placed in a number wheel as shown in the diagram to the right, so that the position after 32767 is not 32768 but is –32768. Similarly (–32768 – 1) is not equal to –32769 but 32767. The following program will clarify the point.

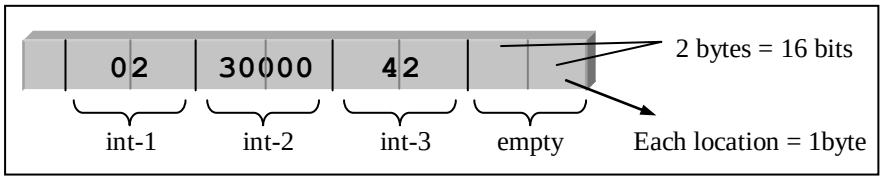



```
/*Program to find the range of integer numbers*/
#include <stdio.h>
void main()
{
```

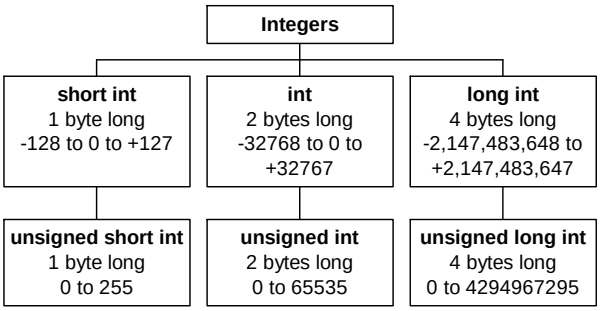
```
int number;
number = 32767;
     printf("\nNumber is: %d", number);
     printf("\nNumber plus 1 is: %d", number+1);
     printf("\nNumber plus 2 is: %d",number+2);
     printf("\nNegative number is: %d",-number);
     printf("\nNegative number minus 1 is: %d",-(number+1));
     printf("\nNegative number minus 2 is: %d",-(number+2));

}
```

The output of the above program will be as shown below:

```
Number is: 32767
Number plus 1 is:-32768
Number plus 2 is:-32767
Negative number is:-32767
Negative number minus 1 is:-32768
Negative number minus 2 is:32767
```



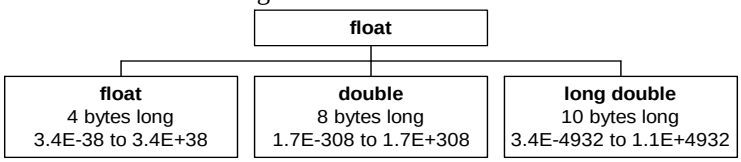

The figure above shows how memory is allotted to a normal integer number. If each small box represents one byte of memory i.e. 8 bits then two such boxes are taken together to store the integer data. Here int-1 stores 2, int-2 stores 30000 and int-3 stores the number 42. In case of higher or lower sizes, more or less memory blocks are set aside to store the required data. Thus depending upon the size of an integer they are further classified into the following types:




Thus an integer can be one, two or four bytes long and further can be **signed** or **unsigned** depending upon the requirement of the problem. Where integers are used for counting purposes there is no need for signed integers (let us assume negative counting is not considered) whereas when considering the result of integer subtraction we have to consider signed integers to take into consideration negative results also.

- **Floating point** type Data: represented as **float (4 bytes long):**

Fractional numbers i.e. numbers **with a value after the decimal point also**, are called floating-point numbers. Simple floating-point numbers are 2.54, 6.023, 13.6 etc. Floating-point numbers can be extremely large or small and hence are often expressed as exponential numbers. For example the number 2.9E+19 is read as 2.9 with an exponent of +19 and indicates that move the decimal point to the *right* by 19 places by adding adequate numbers of zeroes to get the actual number. In algebraic notation the number is represented as $2.9 \times 10^{19}$. Thus the above number is equal to: 29,000,000,000,000,000,000. Similarly a fractional number can be expressed as 5.235E-8, which states that one should shift the decimal point to the *left* by 8 places to get the actual number. Thus the actual number is equal to: 0.00000005235 and its algebraic equivalent is $5.235 \times 10^{-8}$. Accordingly the Avogadro number is represented as 6.023E+23. For positive exponential values the "+" sign can be dropped and the number can be simply written as 6.023E23. In general a floating-point number is 4 bytes long but just like integer variables the different types of **`float`** variables are given below:



As floating-point variables make an approximation, there is limit to the precision of floating point numbers. For example though the number 5.21342345671215 is well within the range of a float but it may be stored as 5.213423. This is called a single precision float type, which means that it is limited to 5 or 6 decimal places. For greater precision a double type floating point number should be taken, which has a double precision and can be accurate up to 14 to 15 digits after the decimal.

- **Character** type Data: represented as **char (1 byte long):**

This type of data represents a **single letter**, numeral or other keyboard characters. For each piece of computer type data the computer reserves just one byte of space. Each char type data has an equivalent integer representation varying from 0 to 255.

Characters thus include the 26 uppercase and 26 lowercase letters, the 10 numeric digits and other punctuations and symbols that can be entered from the keyboard. Character type data can be used where the input required is in the form of a single character like selection from a set of multiple choices. We give below the different characters available as char type data:

| | |
|---|---|
| Uppercase alphabets | **A, B, C, D, E,..... , X, Y, Z** |
| Lowercase alphabets | **a, b, c, d,..... , x, y, z** |
| Digits | **0, 1, 2, 3, 4, 5, 6, 7, 8, 9** |
| Special symbols | **~ ! @ # $ % ^ & * ( ) _ + = - \| { } [ ] \ ″ : ′ ; < > , . ? /** |

Note: Since character type data uses only one byte of memory space hence when numeric data is not used for mathematical purposes it can be declared as char type data and can save one byte of memory.

- **String** type Data:

A string is usually a **group of characters** such as a word, phrase or sentence. C does not have a string data type but treats a string as a character array. More about strings will be discussed later. A string can include any combination of letters, numbers punctuations etc. and is usually represented by placing it inside double quotes like "This is a String". Thus C distinguishes between "123" & 123:- the first one as a combination of characters "1", "2" & "3" and the second one as a number of numerical value 123.

To find out the size of the different data types discussed above the following program can be used which uses the **sizeof()** function to find the length of the data type taken as its argument:

```
/*Program: To Find the Length of Different Data Types*/
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
printf("Program to print the size of Different Data Types\n");
printf("\nThe size of an integer data is (in bytes):%d", sizeof(int));
printf("\nThe size of a short integer data is (in bytes):%d", sizeof(short));
printf("\nThe size of a long integer data is (in bytes):%d", sizeof(long));
printf("\nThe size of a float data is (in bytes):%d", sizeof(float));
printf("\nThe size of a double float data is (in bytes):%d", sizeof(double));
printf("\nThe size of a char data is (in bytes):%d", sizeof(char));
printf("\n\nPress any key to continue");
getch();

}
```

## 1.4 Constants, Variables and Keywords

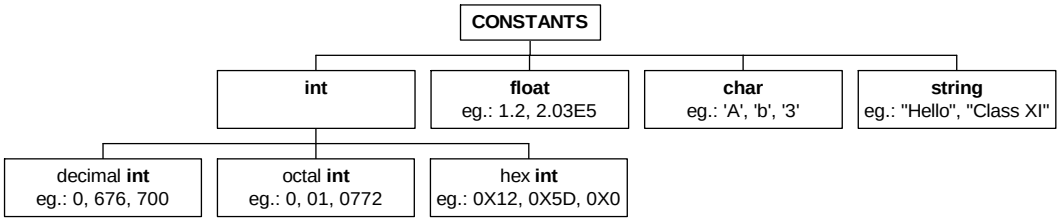
Once we have found out the different basic types of data, we will be now discussing about the nomenclature part of this data, just as a person may be male or female by type but should have a definite name to exactly identify the person.

Each type of data can be broadly divided into a **constant** or a **variable** depending upon the way it is being used in the program. As the name suggests a CONSTANT remains the same or constant throughout the program. The value of a constant is entered during the programming stage and no changes are made to it while running the program. The value of a constant changes only if one changes the program.

Constants are defined when certain fixed data are being used several times in the program. For example if the value of $\pi$ is used several times in a program then it is better to define a constant called PI and store the value in the constant.

In this way one will not have to write every-time 3.1415 wherever the value of π is needed but will simply have to put PI. This saves both time and chances of an error. Moreover if later someone wants to increase the accuracy of PI and make it equal to 3.141592 then he will have to make the change only at one point where the constant is being defined. This saves a lot of time and possible chances of any mistake by missing out on a replacement.

Based on the types of data there are four basic types of constants. The types along with examples are given below.




The rules for constructing the different constants are now given below:

| Type | Rules |
|---|---|
| **Integer** | ▪ It must have at least one digit<br>▪ It **must not** have a decimal point<br>▪ It could be either positive or negative (no sign indicates a positive number)<br>▪ **No** comma or blank should be present within an integer constant<br>▪ The allowable range of an integer constant is from –32768 to +32767 including 0 |
| **Octal Int** | • It must have at least **one digit**<br>▪ It **must have** a **0** before the number |
| **Hex Int** | • It must have at least **one digit**<br>• It **must have** a **0X** or **0x** before the number |
| **Float** | • It must have at least **one digit**<br>• It **must have** a **decimal point**<br>• It could be either **positive** or **negative** (no sign indicates a positive number)<br>• **No comma or blank** should be present within an integer constant<br>• It can be written in either a **fractional form** or as an **exponential form**<br>• When expressed in exponential form,<br>o The **mantissa** part and the **exponential** part should be separated by the letter **e** or **E**<br>o The mantissa part may have a **positive** or **negative** sign<br>o The **exponent** must have at least one **+ve** or **–ve** integer (default is positive)<br>• **Range** of real constants can be from **–3.4e38 to 3.4e38** |
| **Character** | • It can be either a **single alphabet**, single **digit**, or a single **special symbol**<br>• Each constant should be enclosed within **single quotes** like **'A'**<br>• The maximum length of a character constant is one character |
| **Strings** | • It can be **any combination of characters**<br>• Each constant should be enclosed within **double quotes** like "This is a String" |

We now give below a chart of valid and invalid constants of the different variable types, constructed by taking into consideration the rules that have been defined above.

| **Integer** | | **Octal Integer** | | **Hex Integer** | | **Float** | | **Character** | |
|---|---|---|---|---|---|---|---|---|---|
| **Valid** | **Invalid** | **Valid** | **Invalid** | **Valid** | **Invalid** | **Valid** | **Invalid** | **Valid** | **Invalid** |
| **32757** | 32,757 | **096** | 29 | **0X2A5** | 5,201.32 | **+56.23** | 5 | **'A'** | Hi |
| **5** | 32.057 | **02** | -0.2e+2 | **0X5** | 0.0 | **0.0** | 5,201.32 | **'z'** | A |
| **0** | 0.0 | **0** | 0.0 | **0x0** | 010 | **2.0** | 0 | **'b'** | 23 |
| **-10** | 2-9 | | 019 | **0X** | 0F | **-3.2e-5** | | **'?'** | 2e5 |
| **-258** | 2E5 | | 08 | | -0.9E+6 | **-0.2E+2** | | **'4'** | 'char' |

Now that we have discussed about constants let us talk about **variables**. A variable is an identifier that is used to represent some specified type of information that may change in its value or content during the course of running the program. Thus *a variable holds or stores a value that may vary depending upon the input to a program*.

Just like constants, variables also need to be **declared** so that C can assign a portion of the memory to store the variable and to declare a variable a name has to be given to the variable. By declaring a variable and giving it a name, C reserves a space in the memory (e.g.: 2bytes for an int, 4bytes for a float etc.) to store the contents of the variable. The variable should be defined before the program accesses it and thereafter the data item can be accessed simply by referring to the variable name. Thus a variable to store the name of a person can be simply called **`name`** whereas a variable to store the age of a person can be called **`age`**. If a variable called **`age,`** of type **`int`** is declared, then C will reserve 2 bytes of memory location to store the value that will be assigned to **`age`**. However, though a variable has a variable content depending upon the input, but still it can be assigned an initial starting value as will be shown later.

The type of a variable is similar to that of a constant and can be an **int**, **float**, **char** or **string**. As such no further discussion is done in that respect.

**Keywords** are *reserved words* or words that have **already been defined** to the C compiler. These keywords cannot be used to define a constant or a variable as they carry a **predefined meaning** and hence cannot carry a double meaning. Keywords are written in lowercase and there are about 32 keywords that are available in C. They are listed below for reference:

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

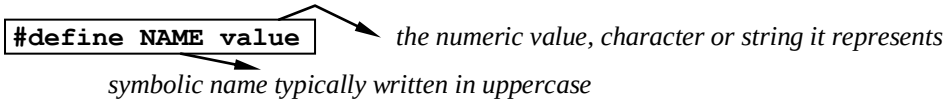
Now let us find out how to declare a constant or a variable i.e. how to let C know which data is a Constant and which one is a Variable.

## 1.5 Declaring Constants & Variables

Declaring a Constant (also called symbolic constant) means telling the C compiler the constant's name and value. All constants must be declared before they can appear in an executable statement. The name of a constant can be anything (excepting the reserved 'C' keywords that we will be discussing later) and can be any combination of uppercase or lowercase letters, digits and the underscore mark ' _ ' but no blanks. **However each name should begin with a letter only**. Names are **case sensitive** i.e. *NAME* and *name* are different constants in C.

While using a symbolic constant the constant name should appear at the places where the value (numerical or string) indicated by the constant is required. During compilation, before the compiler starts to create the object code, it substitutes each occurrence of the constant name with its value.

A constant is defined **before** the **`void main()`** function using the **`#define`** directive. The structure is:




*Note* that a symbolic constant definition **does not end with a semicolon**, as this is not a true C statement. We now give below some examples of valid & invalid constant names and declarations.

| Valid Constant Names | | Invalid Constant Names |
|---|---|---|
| PI | CONST_1 | 555 |
| RATE | First_Lane_Name | 29TH_YEAR |
| NUMBER1 | _Account_No_1 | DATE-OF-BIRTH |
| NUMBER2 | _DATE_OF_BIRTH | "ACCOUNT NUMBER" |
| Table1 | _20TH_NAME | Hello! |
| Table2 | | |

```
#define PI 3.1516          : Defining a float type constant
#define RATE 12            : Defining a int type constant
#define NUMBER 0.5         : Defining a float type constant
#define OPTION1 'A'        : Defining a char type constant
#define REPLY1 "Yes"       : Defining a string type constant
#define ASK "Press Any Key to Continue"  : Defining a string type constant
```

Thus **`#define PI 3.1516`** creates a constant called **`PI`** and gives it the value of **`3.1416`** and wherever the compiler sees **`PI`** in the source code it substitutes the value **`3.1416`** there.

The data type need not be stated explicitly when defining a constant. 'C' automatically assigns a data type based on the value given in the **`#define`** directive. Thus:

**`#define RATE 12`** → will be assigned a **int** data type because 12 is an integer &
**`#define PI 3.1516`** → will be assigned a **float** data type as 3.1516 is a float while
**`#define REPLY1 'A'`** → will be assigned a **char** data as 'A' is a char type data.

**Example**: Calculate the equatorial-circumference, cross-sectional-area, surface-area and volume of a sphere of radius r.

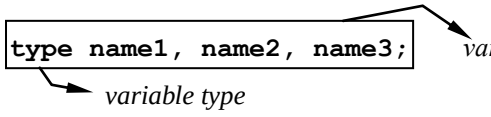
```
/*Program to find circumference, area & volume of a Sphere of radius r*/
#include<stdio.h>
#define PI 3.1416
void main()
{
int radius;
float circum, area, surf_area, vol;
printf("\nInput the Radius (integer number): ");
scanf("%d", &radius);
circum = 2*PI*r;
area = PI*r*r;
surf_area = 4*PI*r*r;
vol = (4.0/3)*PI*r*r*r;
printf("\nRadius=%d, \nCircumference=%f, \nArea=%f,", radius, circum, area);
printf("\nSurface Area=%f, \nVolume=%f", surf_area, vol);

}
```

Let us now discuss how to **declare a variable**. Variable names are usually written in *small letters* and all the rules that are true in naming a constant are true for variables also. Thus examples of variables include:

```
total
month1
_5th_grade_marks
tax_rate
```

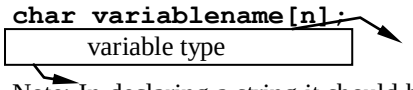
***Note:*** Excepting '_' no other punctuation is allowed in naming a variable and a variable should not start with a numeric digit.

A **declaration** of a variable consists of a data type followed by one or more variable names ending with a semicolon. A single variable name or several variables of the same type can be declared within a single instruction with each variable name separated by a comma. e.g.

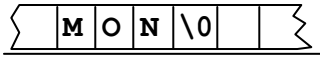



**`int count, days;`** → 2 integer type variable called count & days
**`float area, rate;`** → 2 float type variables called area and rate
**`char initial;`** → 1 char type variable called initial

Let us now find out how to define a string variable. C does not contain any string type and allows to work with string type data through the use of arrays. Since a string can be defined as a series of characters, it can be treated as a series of char type variables collected into a group called an array. The characters of the string are stored together. Each character is stored in a separate memory location just like a single character, but can be displayed like a single string in one **`puts()`** command. The symbol **`\0`** is a special code in C called a **null terminator** and is inserted at the end of a string to *mark the end of the string*. Thus the char type variable is used to declare a string in the following manner:




Note: In declaring a string it should be kept in mind that the number in the square bracket should be 1 more than the maximum number of characters that one wants to store, to accommodate the null terminator **`\0`**. e.g. **`char days[4];`**




The figure to the left shows how the above string declared as **`days[4]`** is stored in the

memory. It can be seen that of the 4 places, 3 are used to store the characters MON and the 4th one is used to store the null terminator **\0**. A string variable will hold only the number of characters it has been assigned for and will show a runtime error if one enters more than that amount.

To make a *user defined name for a variable type* C provides us with the **typedef** command. It is used to redefine the name of an existing variable type. The general syntax of typedef is:

```
typedef existing-variable-type-name  new-variable-type-name;
```

e.g.:
```
typedef unsigned long int TREE;
TREE teak, mahogany, mango;
```
In the above example instead of writing "*unsigned long int teak, mahogany, mango;*" a more sensible new variable type name called TREE is defined and assigned to *unsigned long int* to declare variables called teak, mahogany, & mango (which may be used to store the count of such trees). Usually the new type name is written in *capital letters* to make it clear that we are dealing with a renamed data type.

## 1.6 Assigning initial values to variables

A **literal** is any piece of **data** that one **directly types in** into a C program. It can be any number, character or string that one may use as an initial value in the program. Though a constant literal remains the same throughout the program but a literal assigned to a variable may change in course of a program. Thus:

```
#define PI 3.1415;          → Here 3.1415 is a Constant literal
rate=12.5;                  → Here 12.5 is a float variable literal
puts("Welcome to C");       → Here "Welcome to C" is a string literal
```

Variables may have an initial value i.e. one may want a variable to have a certain value at the start of a program though this value can change as the program runs. This initial value can be assigned either as a declaration when the variable type is declared or as a separate instruction. Thus:

Ex.①
```
int age=10, count=5, number;
char initial= 'A';
```
Ex.②
```
int age, count, number;
char initial;
age=10;
count=5;
initial= 'A';
```
Both example ① and ② perform the same job i.e. declare three **int** type variables called *age*, *count*, *number* and assign initial values to *age* and *count* and declare a **char** type variable and assign it a value **'A'**. In the first case the initial value is given as a part of the declaration whereas in the second case they are declared first and then given the initial values through separate instructions. No initial value is given to *number* in both the examples. **Note**: a **char** type literal should be put between two single quotes **''**.
When dealing with **strings**, once a string variable has been declared one cannot directly assign an initial value to the string like a **char** or numeric variable. Hence the statement:

```
char client[16];
client = "Subhas Bose";
```
} is **not** a valid statement

A string can be assigned an initial value while declaring it, in the following ways:

① `char client[]= "Subhas Bose";`
② `char client[12]= "Subhas Bose";`

Here the variable **client[]** will take 11 characters only and the computer reserves 12 character spaces for the variable **client[]** (including the null terminator **\0**). But care should be taken while declaring the string as in the second case, as the number within the square brackets should tally with the number of characters in the string (i.e. should be one more than the character content to accommodate the null terminator **\0**). In case there is a mismatch, either a truncation will happen or meaningless characters will be introduced in the extra spaces.

## 1.7 Operators: some simple operators in C

After an input is made, it needs to be processed to give any useful information as output. This processing of data may involve mathematical operations. **Operators are required to perform the calculations** that transform data into information. It is basically a symbol that tells the computer how to process the data. Based on their mode of operation, operators can be of four types viz. **arithmetic**, **increment**, **assignment** & **logical**. At present we will discuss the arithmetic & assignment operators and the rest will be discussed later. The following operators perform the mathematical functions as indicated beside their names:

= → **assignment**, e.g. y = 5 or y = a+b-c i.e. used to assign a value to a variable
\+ → **addition**, e.g. y = a+b → for a=8, b=5, y=13
– → **subtraction**, e.g. y = a-b → for a=8, b=5, y=3
***** → **multiplication**, e.g. y = a*b → for a=8, b=5, y=40
/ → **division**, e.g. y = a/b → for a=8, b=5, y=1 (a & b being integers the answer will be an integer also and not 1.6 which is a float)
% → **remainder**, e.g. y = a%b → for a=8, b=5, y=3 (i.e. the remainder when dividing 8 by 5)

When using an operator to perform mathematical calculations, a variable is usually used to store the result of the calculation. The variable and the arithmetic operation are connected by an equal sign "=" also called the assignment operator, with the variable on the left side and the calculation on the right side. There can be any number of variables connected by the operators on the right side of the "=" sign, but on the left there should be only the name of the variable. E.g.

`area = 3.1415*r*r;` → Calculates the area of a circle by multyplying the constant `3.1415` with `r*r` (**Note: <u>there is no exponential operator "^" in C</u>**)
`interest =(p*r*t)/100` → Finds the interest by multiplying p, r & t and then dividing by 100

Let us now find out the order in which C performs the calculations. The rules are the same that you have learnt in your junior classes i.e. BODMAS with the inclusion of the modular division operator `%` i.e.:

**{B**racket**}**→**{M**ultiplication-**D**ivision-**Mo**dular**D**ivision**}**→**{A**ddition-**S**ubtraction**}**→**{**Equal**}**

*Priority*: 1st → 2nd → 3rd → 4th

The above rule depicts the hierarchy of operations with more than one operator. Of the above, the operators put inside the { } share the same hierarchy i.e. Multiplication-Division-ModularDivision share the same priority i.e. any one of them can be performed before the other. Similar priority is shared by Addition-Subtraction also. Thus 20*3/5 is the same as 20/5*3 and is equal to 12.

The following examples will clarify the use & priority of the different operators.

**Example1:** Find the value of the expression: $b^2-4a[3a^2\{(7b^2-5)\div 2\}-35200]-5$ for a=2 & b=29

**Solution**: Do carry out the above calculation we will have to first convert the above expression to the syntax of C. In algebra we read ***4a*** as ***4 times a*** or **4x<u>a</u>**. These assumptions we cannot make when writing a C program and we have to explicitly put the different operators where they are needed. Moreover we can only use the parentheses `()` inside an arithmetic calculation and <u>not any other braces</u> like `{}` or `[]`. Hence the above expression becomes:

```
b*b-4*a*(3*a*a*((7*b*b-5)/2)-35200)-5
```


We can see that "*" has been inserted at the places wherever there is a multiplication, the slash "/" has been inserted for a division will all the brackets changed to `()`. Let us now see the order in which this is calculated for the values a=2 & b=29:

Step1: `29*29-4*2*(3*2*2*((7*29*29-5)/2)-35200)-5`
Step2: `841-8*(12*((5887-5)/2)-35200)-5`
Step3: `841-8*(12*5882/2-35200)-5`
Step3: `841-8*(35292-35200)-5`
Step4: `841-8*92-5`
Step5: `841-736-5`
Step6: `100`

**Example2**: Write a program to find how much is to be added to make an integer exactly divisible by 5 and find the new integer.
**Solution**: First let us find out the algorithm and then do the calculation as per the algorithm.
*Algorithm*:
Input the number
Divide the number by five
Find the remainder
Subtract the remainder from 5 to get the required number to be added
Add this result to the input number to get the new integer

```
/*Program to find number divisible by 5*/
#include<stdio.h>
#include<conio.h>
void main()
{
int num, newnum;
clrscr();
printf("\nInput number to check divisibility by 5: ");
scanf("%d",&num);
newnum=num+5-num%5;
printf("\nThe next highest number to %d, divisible by 5 is %d", num, newnum);
printf("\nPress any key to exit\n");
getch();

}
```

In the above example the calculation for **newnum** is done in the following order for num = 18:

```
newnum = num+5-(num%5)   → First modulo calculation is done
       = 18+5-(18%5)
       = (18+5)-3        → Second addition is done
       = 23-3            →Finally subtraction is done
       = 20              →To get the correct number 20
```

If priority of operators was not maintained the result would have been:

```
newnum = 18+5-18%5
       = 23-18%5
       = 5%5
       = 0               → Which is the wrong answer
```

## 1.8 Operators & Data Types

Usually the same type of data is used on either side of the equal sign. Thus when adding two *float* numbers, the result is assigned to a *float* variable. But in case there are different numeric types on the left and right sides of the assignment operator, the value displayed depends on the type of variable on the left side of the equal sign. Thus when dividing an integer with another integer, if the result is assigned to an `int`, then the output will be an integer irrespective of whether the dividend is a multiple of the divisor or not. The result will show only the integer portion of the result without any decimal part. For example:

```
int dividend, divisor, result;          → Declares 3 int type variables
printf("Enter Dividend, Divisor");
scanf("%d,%d", &dividend, &divisor);    → Inputs the numbers say dividend =23, divisor =5
result = dividend/divisor;              → Performs the division and gets result=4 (not 4.6)
printf("Result is %d", result);         → Displays the result as an integer i.e. 4
```

To rectify the above problem we have to declare **result** as a **float**. Thus when there is a mismatch of variable types then the value of the expression is demoted or promoted depending on the type of variable on the left hand side of the = sign. Therefore:

```
int i; float f;     i.e. i is defined as an integer & f as a float
i=4.9               Here i will hold the value of 4 i.e. it gets demoted from a float i.e. 4.9 to 4
f=6                 Here f will hold 6.000000 i.e. int 6 gets promoted to a float 6.000000
```

The same logic holds for complex calculations also, i.e. in the following program piece:

```
float a, b, c; int root1;
root1 = (1/(2*a))*(-b+sqrt(b*b-4*a*c));
```

The root1 will be an integer though in the above assignment statement there are both integers and floats.

Also note that the maths calculations are performed before converting the data types and then the assignment is completed. Thus in the following example though the result is obtained as an **int** but since two **float** numbers are added, the additions performed as a **float** and while displaying the result the decimal part is truncated and only the integer part is displayed.

```
void main()
{
int total;
float cost=45.09, shipping=9.92;
total=cost+shipping;        → total = 45.09+9.92 = 55.01 = 55 (since total is an int)
printf("\nThe total bill amount is Rs. %d", total);

}
```

In the above example the calculation is done with the **float** variables 45.09 & 9.92 but since the variable **total** is an **int** thus the value is first converted and then displayed as an integer i.e. 55. Had the calculation was been done first by converting the data into integers then the result would have been (45+9)=54 and not 55.

**Important Note1**: When dividing two **literal values** and assigning the result to a float, **at least one must have a decimal place** if the result is to be displayed as an output with decimal places i.e. as a float. The program below demonstrates the fact by doing the same calculation four times and observing the result.

```
void main()
{
float c;
c=12/5;      printf("%f\n",c);          → Output is 2.000000
c=12.0/5;    printf("%f\n",c);          → Output is 2.400000
c=12/5.0;    printf("%f\n",c);          → Output is 2.400000
c=12.0/5.0;  printf("%f\n",c);          → Output is 2.400000

}
```

**Important Note2**: When dividing two **variables**, if it is known that the output can be a **float**, then apart from declaring the output as a float, **any one of the variables has to be declared as a float** (i.e. either the divisor or the dividend) to get the correct answer. The following programs display the different combinations of inputs and the resulting outputs depending upon the declaration type.

- **Case1: integer Dividend, Divisor, Quotient**:

```
#include<stdio.h>
void main()
{
int a,b,c;
printf("Input number1: "); scanf("%d",&a);
printf("Input number2: "); scanf("%d",&b);
c=a/b;
printf("The value of %d by %d is %d",a,b,c);

}
```

**Output:**

```
Input number1: 4
Input number2: 8
The value of 4 by 8 is 0
```

- **Case2: integer Dividend, Divisor: float Quotient**:

```
#include<stdio.h>
void main()
{
int a, b; float c;
printf("Input number1: "); scanf("%d",&a);
printf("Input number2: "); scanf("%d",&b);
c=a/b;
printf("The value of %d by %d is %f",a,b,c);

}
```

**Output:**

```
Input number1: 4
Input number2: 8
The value of 4 by 8 is 0.000000
```

- **Case3: integer Dividend: float Divisor, Quotient**:

```
#include<stdio.h>
void main()
{
int a; float b, c;
printf("Input number1: "); scanf("%d",&a);
printf("Input number2: "); scanf("%f",&b);
c=a/b;
printf("The value of %d by %f is %f",a,b,c);

}
```

**Output:**

```
Input number1: 4
Input number2: 8
The value of 4 by 8.000000 is 0.500000
```

- **Case4: integer Divisor: float Dividend, Quotient**:

```
#include<stdio.h>
void main ()
{
int b; float a, c;
printf("Input number1: "); scanf("%f",&a);
printf("Input number2: "); scanf("%d",&b);
c=a/b;
printf("The value of %f by %d is %f",a,b,c);

}
```

**Output:**

```
Input number1: 4
Input number2: 8
The value of 4.000000 by 8 is 0.500000
```