# C++ PRELIMINARIES

**TOKENS** : As we know the smallest individual units in a program are known as tokens. C++ has the following tokens-

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using tokens, white spaces and the syntax of the language.

**KEYWORDS** - The keyword implement specific C++ language features. They are explicitly reserved identifiers and can't be used as names for the program variables or other user defined program elements. Some of the C++ keywords are illustrated; which are in addition to set of ANSI C keywords.

| | | | |
|---|---|---|---|
| asm | private | protected | new |
| friend | delete | operator | inline |

……..etc.

**IDENTIFIERS**- identifiers refers to the names of variables, functions, arrays, classes etc. created by programmers. These are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common : ( both in C, C++ )

- Only alphabetic characters, digits and underscores are permitted
- The name can't start with a digit
- Uppercase and lowercase letters are distinct
- A declared keyword can't be used as a variable name.

However major difference lies in the limit of the length of a name. While ANSI C recognizes only first 32-characters in a name, C++ places no limits on its length and therefore, all the characters are significant.

**BASIC DATATYPE** – C++ compilers support most of the basic data type supported by ANSI C compilers namely **int, char, float, double, void**. With void being only exception most of the basic data type have several modifiers namely **long, short, signed, unsigned**.

The purpose of the data type **void** is contrasting. These includes

i) specifications of return type of a function not returning any value.
ii) to indicate an empty argument list to a function.
iii) the introduction of generic pointers. E.g.

```
                    void   *gp;
```

A ***generic pointer*** can be assigned a pointer value of any basic data type , but, it may not be de-referenced. E.g.

```
                    int   *ip;
          gp = ip                // assigns int pointer to void pointer
```

is a valid statement.

```
          *gp = *ip
```

is not a valid statement for it won't make sense to de-reference a pointer to a void value.

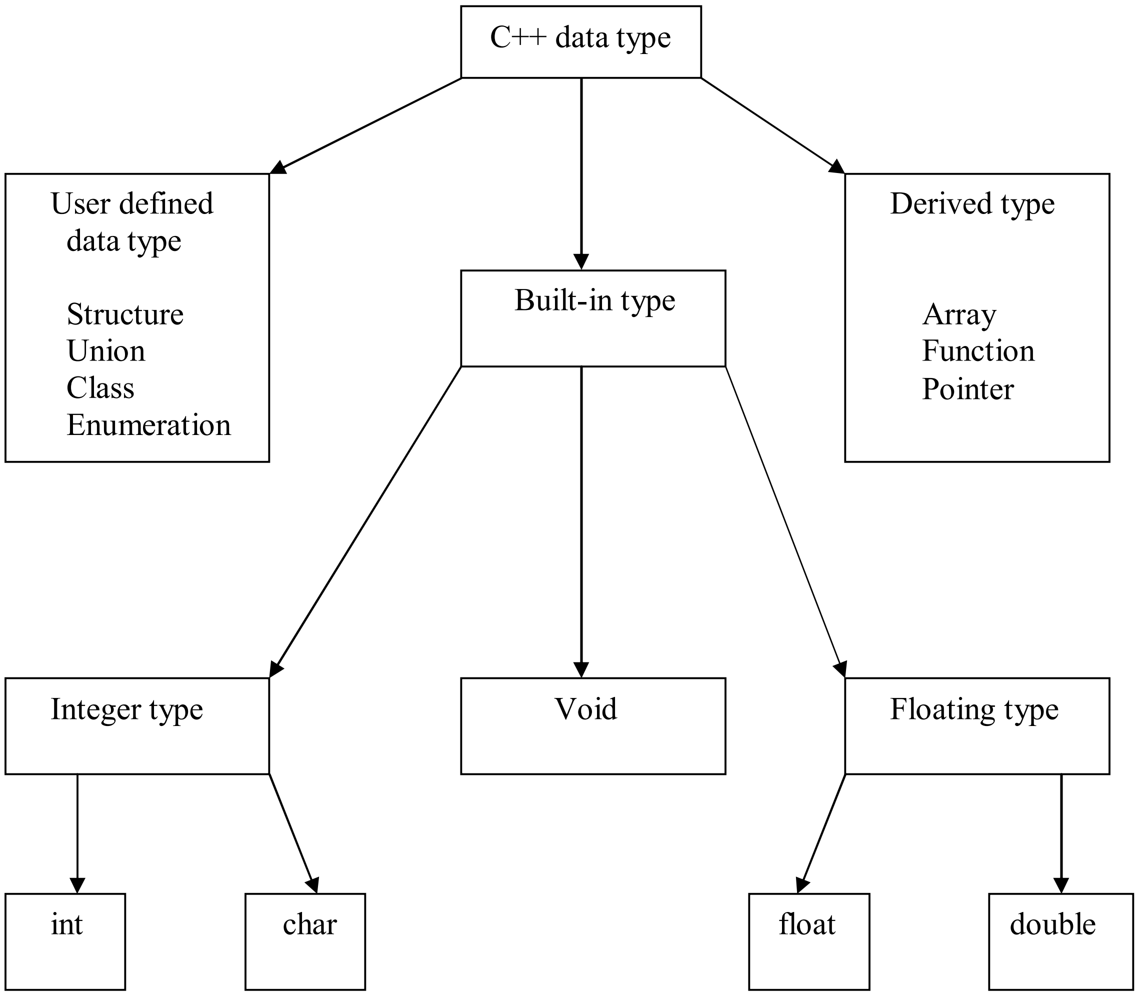
Assigning any pointer type to a void pointer without using a cast is allowed in both C++ and ANSI C. in ANSI C assigning a void pointer to a non-void pointer was permissible. i.e.

```
void    *gp;
char    *ip;
ip=gp;
```

A void pointer can't be directly assigned to other type pointer in C++.
We need to use a cast operator as shown below

```
ip= ( int  * ) gp;
```

Following figure illustrate various data type supported by C++



**USER DEFINED DATATYPE**

<u>Structure & Classes</u>

we have used user-defined data-type structure & union in C. in addition to that C++ also permits to define another user-defined data type known as class, that can be used to declare variables. Class variables are referred to as objects.

**<u>Enumeration</u>**

An enumerated data type is another user defined data type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The enum keyword (from C) automatically enumerates a list of words by assigning them values 0, 1, 2 and so on.

The syntax of an enum statement is similar to that of struct statement.

e.g.

```
        enum  discipline { B. Tech, BCA, MCA }
        enum  department { ECE, EE, CSE }
```

Enumerated data type differs slightly in C++ for tag names discipline & department becomes new data type. We can declare new variable using these tag names

```
        discipline  BBA;
        department  IT;    ..etc.
```

C++ doesn't permit an int value to be automatically converted to an enum value i.e. in C++, each enumerated data type retains its own separate type.

By default, the enumerators are assigned integer value starting with 0 for the first enumerator, 1 for the second & so on.

We can over-ride default by explicitly assigning integer values to the enumerators.

e.g.

```
        enum     color { red, blue=4, green= 8 }
        enum     color { red=5, blue, green }   are valid statement
```

In the first case red is 0 by default. In the second case blue=6, green=7.

i.e. subsequent initialized enumerators are larger by one than there predecessors.

Program :

```
            enum   shape
            {
                    circle, rectangle, triangle;
            };
            main( )
            {
                    cout<< " Enter shape code:";
                    int  code;
                    do
                    {
                            cout<<"Enter shape code:";
                            cin>> code;

                            switch(code)
                            {
                                    case      circle :  ……….;
```

```
                                                  break;

                                   case rectangle :  ……….;
                                                  break;

                                   case  triangle  :  ……….;
                                                  break;
                              }
                         } while(code>=circle && code<=triangle);
                    }        /*  end main   */
```

**DERIVED DATATYPE**

Arrays
The application of arrays in C++ is similar to that of C only exception being the way the character array are initialized. C compiler allow us to declare array size as the exact length of the string constant . i.e.

char   string[3]= "XYZ"   is valid in ANSI C.

it implies that the null character '\0' will be left out.
But in C++ it is not permissible i.e. size should be one larger than the number of character in the string.

Functions
Function has undergone major change over in C++. While some of this changes are  simple, others require a new way of thinking when organizing our program. Many of these modifications and improvements were driven by the requirement of the object-oriented concept in C++. Some of these were introduced to make C++ program more readable & reliable.

Pointers
Pointers are declared & initialized as in C

```
          int  x;
          int *pi;
          pi=&x;  etc.
```

C++ adds the concept of ***constant pointer*** **&** ***pointer to constant*** .
e.g.          char *const ptr1 = "NEW"          // constant pointer
we can't modify the address that ptr1 is initialized to.

Consider      int const   *ptr2=&m;          //pointer to a constant
ptr2 is declared here as a pointer to a constant. It can point to any variable of permissible type, but content of what it points can't be changed.
We can declare both the pointer & the variable as constants in the following way :

const   char  * const  cp="XYZ".

In this case neither the address assigned to cp nor its content can be changed.
Pointers are extensively used in C++ for polymorphism & memory management.

**Symbolic Constant**

There are two ways of creating symbolic constants in C++

1. Using the quantifier const.
2. By the keyword enum.

In both C and C++ , any value declared as const can't be modified by the program. However there are some differences in implementation in C++, we can use const in a constant expression such as

```
const int size = 10;
char name[SIZE];
```

This would be illegal in C. This used instead of #define to create constant having no type information.
Const produces int by default in C++. i.e.

```
        const  size=10;
    means
        const int size=10;
```

the following features corresponding to constant may observed

- ANSI C doesn't require but C++ compilers require const to be initialized. Default initialized value is zero.
- The scooping of const value differs. A const in C++ defaults to the internal linkage and hence it is local to the file. In ANSI C const values are global in nature.
- Enum { X, Y, Z }; is another method of defining symbolic constants.

**Type compatibility**

C++ is very strict with regard to type compatibility. C++ defines int, long int, short int as different data type. Type casting is required when their values are assigned to one another. Similarly signed char, unsigned char, char are considered as different data type although each occupies single byte.
In C++, the type of values must be the same for a complete compatibility. Otherwise, a cast must be applied.
These restrictions in C++ are necessary in order to support **function overloading** where the two functions having the same name are distinguished using the type of function arguments.
Another difference is the way char constants are stored . In C, they are stored as 'int'. therefore sizeof('X') is equivalent to sizeof(int).
But in C++, sizeof('X') equals to the sizeof(char) for char is not promoted to sizeof(int).

**Flexible declaration**

The general proposition that all variables must be declared before they are used in executable statement holds good in C++ too.

Otherwise there is a significant difference between C and C++ with regard to the place of the declaration.
C requires all the variables to be declared at the beginning of the scope(block). We come across a group of variable declarations at the beginning of each scope level. Their actual use might be far away from the scope. C++ make the programming job much easier and scope of error free by allowing declaration any where in the scope i.e. a variable can be declared right at the place of its first use.

```
        Int main( void)
        {
                float  x;
                ……..;
                for( int  i=1;I<5;I++)
                {
                        cin>>x;
                        sum+=x;
                }
        }
```

**Dynamic Initialization**

C++ permits initialization of the variables at  run-time. This is referred to as dynamic initialization. In C variables must be initialized using a const expression and the C compiler would fix the initialization code at the time of compilation.
In C++ a variable can be initialized at run-time using expressions at the place of declaration. E.g.

```
                        int  n = strlen ( string );
                        flaot   area = 3.14159 * rad *rad;
```

So declaration and initialization may be combined as

```
                        float   average = sum / i ;   // initialize dynamically at run-time
```

**Reference variable**

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alternative name (alias) for a previously defined variable. The general syntax to declare reference variable is

```
        datatype   &   reference_varname = variable_name
e.g.             flaot   total = 100;
                 float  & sum= total;
```

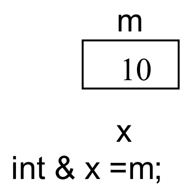
here sum is the alternative name declared to represent the variable total and they can be used interchangeably. A reference variable must be declared with initialization. Note that initialization to a reference variable is completely different from assignment to it.

Both the variable and its alias refer are allocated the same location in the memory.

**Call by reference**

A major application of reference variable is in passing arguments to function.

```
void  f(int & x)
{
        x+=10;
}
main( )
{
        int  m=10;
        f(m);
}
```



when the function call f(m) is executed, the following initialization occurs :

```
int & x =m;
```

thus  x  becomes alias of m after executing call f(m) till the control remains in the user defined function.
Call by reference is a useful mechanism in OOP that it permits manipulation of the object by reference.

**Implicit conversions**
C++ allows mixing of variables datatype i.e. integral expression, float expression pointer expression etc.
e.g.  m = 5 +2.75 is a valid statement.
Wherever datatype are mixed in an expression, C++ performs the conversions automatically. This process is known as implicit or automatic conversion.

## OPERATORS

C++ has a rich set of operators. All C operators are valid in C++ also. Moreover C++ introduces some new operators. Two of such are

| | |
|---|---|
| << | insertion or put- to |
| >> | extraction or get-from |

apart from them there are

| | |
|---|---|
| : : | scope resolution operator |
| : :* | pointer to member declarator |
| .* | pointer to member access operator |
| ->* | member dereferencing operator |
| new | memory allocation operator |
| delete | memory release operator |
| endl | line feed operator |
| setw | field width operator |

last two often referred to as manipulator.

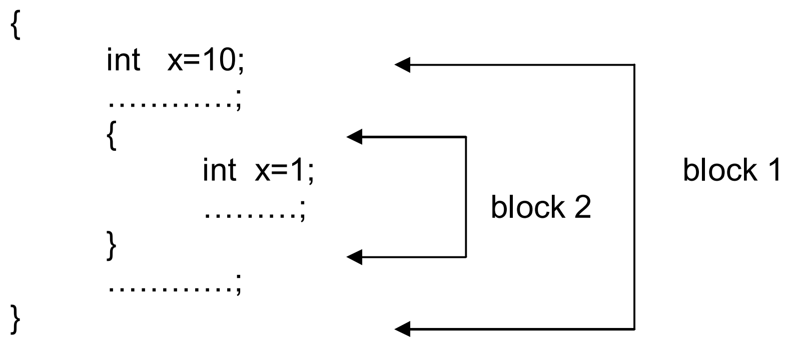
In addition C++ allows us to provide new definition to some of the built-in operators. That is, several meaning can be given to an operator, depending upon the type of argument used. This process is known as **operator overloading.**

**Scope resolution operator**

Like C, C++ is also a block structured language. Same variable can be used in different blocks. The scope of the variables extends within the block where it was declared.
e.g.

```
{
        int   x=10;
        ………..;
}
{
        int   x=1;
        ………..;
}
```

the two different declarations of x refer to two different memory locations containing different values. Statement in the second block can't refer to the variable declared in the first block.
Blocks in C++ are often nested.
e.g.



```
{
        int   x=10;
        ………..;
        {
                int  x=1;
                ………;
        }
        ………..;
}
```

Block 2 contained in Block 1. note that a declaration in the inner block hides the declaration in outer block, therefore, each declaration of x causes it to refer to a different data object. Within the inner block , the variable x will refer to the data object declared therein.
In C global version of a variable can't be accessed from within the inner block.
C++ resolves this problem by introducing a new operator called <u>scope resolution operator</u> ( : : ). This can be used to uncover a hidden variable.

```
: :  variable-name
```

#include< iostream .h>

```
int m=10;
int main(void)
{
        int  m=20;
        cout<<" \n m in inner block :"<<m;
        cout<<"\n uncovered m :"<<: :m;
}
output :
                m  in  inner block  :  20
                uncovered  m :10
```

A major application of the scope resolution operator is to identify the class to which a member function belongs.

**Member dereferencing operator**

Accessing the class members through pointers is allowed in C++. In order to achieve this C++ provides a set of three pointer-to-member operators.

| | |
|---|---|
| : :* | *pointer to member declarator.*<br>to declare a pointer to a member of a class. |
| .* | *pointer to member access operator.*<br>to access a member using object name and a pointer to that member . |
| ->* | *member dereferencing operator.*<br>to access a member using a pointer to the object and a pointer to the member. |

e.g. consider the following class

```
                class  A
                {
                        private :  int   m;
                        public  :  void show( );
                }
```

we can define a pointer to the member m as follows:

```
                        int  A : :*ip = & A ::m
```

the ip pointer thus created ,acts like a class member.
The phrase  A : :* means "pointer-to-member of class A".
The phrase &A : : m  means "address of  m member of class".

Note that int  *ip = &m is illegal for m is not simply an int data type.
The pointer   ip  can be used to access the member m inside member function or friend function.
Suppose that  a  be an object of class A.

```
                        cout<< a.*ip;
                        cout<< a.m;
```

are equivalent.
Suppose ap is a pointer to the object a

```
            i.e.      ap = &a;
```

The operator ->* or dereferencing operator is used to access a member when we use pointers to both member and object.

```
                cout<< ap->m;
                cout<<ap->*ip;
```

are equivalent.

## Memory management operator

C uses malloc( ) and calloc( ) functions to allocate memory dynamically at run-time. Similarly it uses the function free( ) to free dynamically allocated memory.
Apart from supporting above two, C++ also defines two unary operators **new** and **delete** those perform the task of allocating and freeing the memory in a better way.
Since these operators manipulate memory on the free store, they are also known as free store operator.

The new operator allocates sufficient memory to hold a data object with any datatype and returns the address of the object. The datatype may be any valid datatype. The pointer variable holds the address of the memory space allocated .
e.g.

```
                int    *p ;
                float  *q ;
                p = new  int;
                q = new float;
writing     int  *ip = new  int ;     also permissible.
```

General form :

```
pointer variable  = new  datatype ;
```

When a data object is no longer required, it is destroyed to release the memory space for re-use using the delete operator.

The general form :

```
delete  pointer variable ;
```

The pointer variable is the pointer points to a data objects created with new.

```
                delete   p;
                delete   q;
```

In order to delete an entire array pointed by p following statement may be used

```
delete[ ].p ;
```

// p points to an array

previous versions required size of the array to be specified.

**Manipulators**

Manipulators are operators that are used to format the data display. The most commonly used manipulators are 'endl' and 'setw'.

The 'endl' manipulators, when used in an output statement , causes a linefeed to be inserted. It has almost same effect as that of '\n'

e.g.

```
int   m = 7;
cout<< m<<endl;
```

output screen

```
7
=
```

within the printing zone, to make output justified, 'setw' manipulator is used.

The manipulator setw(n) specifies a field width n for printing the value of the variable sum. The value is right justified within the field as shown below –

```
cout<< setw(5)<<m;
```

output screen

| | | | | |
|---|---|---|---|---|
| | | 3 | 4 | 5 |

**Type cast operator**

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Traditional C casts are augmented C++ by a function call notation as a syntactical alternative . the following two versions are equivalent :

```
(type-name) expression          // C notation
type-name ( expression )        // C++ notation
```

e.g. average = float(sum);

⇒ type name behaves as if it is function for converting values to a designated type.

However p = int *(q); is not permissible

In this case the traditional C notation is used.

i.e. p = (int *) q;

otherwise that can be used by means of a typedef statement followed by casting i.e.

```
typedef  int  *  int_pr;
```

```
p = int_pr(q);
```