Unit 2 Tokens, Expression and Control Structures

Tokens : 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 these tokens, white spaces and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.

Keywords : The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user defined program elements.

Identifiers and Constants : Identifiers refer to the name of variables, functions, arrays , classes, etc. created by the programmer. They are the fundamental requirement of any programming language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++ .

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

C++ Keywords

Asm Auto Break case catch char class const continue default delete do double else enum extern float for friend goto if inline int long new operator private protected public register return short signed sizeof static struct switch template this throw try typedef union unsigned virtual void volatile while

Constants : Constants refer to fixed values that do not change during the execution of a program. Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constant do not have memory locations. Example

- 123 // decimal integer
- 12.34 // floating point integer
- 037 // octal integer
- 0X2 // Hexadecimal integer
- “C++” // String Constant
- ‘A’ // Character constant

C++ also recognizes all the backslash character constants available in C.

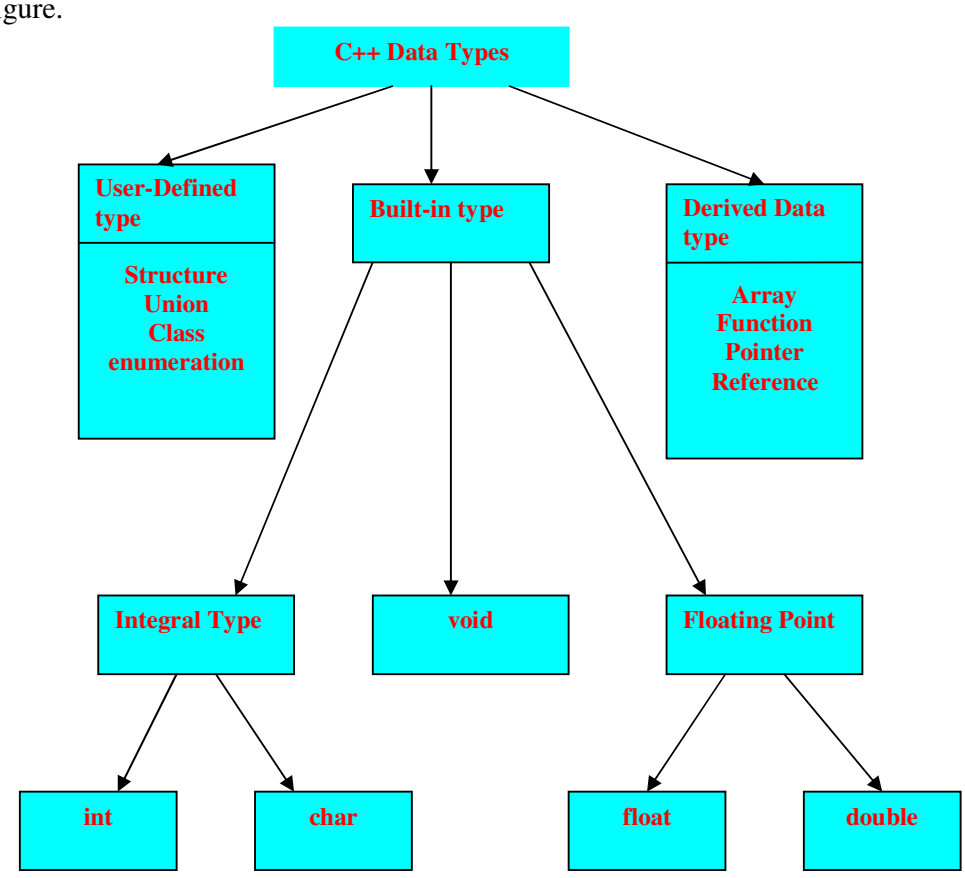
Strings : C++ supports two types of string representation – the C-style character string and the string class type introduced with standard C++. Although the use of the string class is recommended.

Operators : C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. Following is the list of such operators.

- **<< Insertion Operator**
- **>> Extraction Operator**
- **:: Scope Resolution Operator**
- **::* Pointer to member Declarator Operator**
- **->* Pointer to member Operator**
- **.* Pointer to member Operator**
- **delete Memory release Operator**
- **endl Line feed Operator**
- **new Memory Allocation Operator**
- **setw Field width Operator.**

Basic Data Types

Data types in C++ can be classified under various categories as shown in following figure.



- Both C and C++ compliers support all the built-in (also known as basic or fundamental) data types.

- With the exception of **void**, the basic data types may have several modifiers preceding them to serve the needs of various situations.

- The modifiers **signed**, **unsigned, long** and **short** may be applied to character and integer basic data types. However, the modifier long may also be applied to **double.**

- Data type representation is machine specific in C++.

- Table lists all combination of the basic data types and modifiers along with their size and range for a 16-bit machine.

Type	**Bytes**	**Range**
Char	1	-128 to 127
Unsigned char	1	0 to 255
Signed char	1	-128 to 127
Int	2	-32768 to 32767
Unsigned int	2	0 to 65535
Signed int	2	-32768 to 32767
Short int	2	-32768 to 32767
Unsigned short int	2	0 to 65535
Signed short int	2	-32768 to 32767
Long int	4	-2147483648 to 2147483647
Signed long int	4	-2147483648 to 2147483647
Unsigned long int	4	0 to 4294967295
Float	4	3.4E-38 to 3.4 E+308
Double	8	1.7E-308 to 1.7E+308
Long double	10	3.4E-4932 to 1.1E+4932

The type void was introduced in ANSI C. Two normal uses of void are

(1) To specify the return type of a function when it is not returning any value.
(2) To indicate an empty argument list to a function.

```
void funct1(void);
```

User-Defined Data Types

Structure and Classes

User-Defined data types such as struct and union are introduced in C. These data type are legal in C++, another user-defined data type known as class which can be used just like any other basic data type to declare variables. The class variables are known as objects which are the central focus of object-oriented programming.

Enumerated Data Type

An enumerate 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 0, 1, 2 and so on. This facility provides an alternatives means for creating symbolic constants. The syntax of an enum statement is similar to that of the struct statement. For examples :

- **enum shape{circle, square, triangle};**
- **enum colour{red, blue, green, yellow};**
- **enum position{off, on};**

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names shape, colour, and position become new type names. By using these tag names, we can declare new variable. Example :

- **shape ellipse;**
- **colour background**;

ANSI C defines the type of enums to be ints. In C++ each enumerated data types retains its own separate type. This means that C++ does not permit an int value to be automatically converted to an enum value.

Examples :

- **colour background = blue;** **// allowed**
- **colour background = 7;** **// Error in C++**
- **colour background = (color) 7;** **// OK**

However an enumerated value can be used in place of an int value.

int c = red; **// valid**

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by explicitly assigning integer values to the enumerators. For example,

- **enum colour{red, blue = 4, green = 8};**
- **enum colour {red = 5, blue, green};**

are valid definitions. In the first case, red is 0 by default. In the second case, blue is 6 and green is 7. Note the subsequent initialized enumerators are larger by one than their predecessors.

In practice, enumeration is used to define symbolic constants for a switch statement.

Derived data types

Arrays

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as exact length of the string constant. For instance,

```
char string[3] = "xyz";
```

is valid in ANSI C. It assumes that the programmer intends to leave out the null character \0 in the definition. But in C++, the size should be on larger than the number of characters in the string.

Functions

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

Pointers

Pointers are declared and initialized as in C. For examples :

- `int *ip;`
- `ip = &x;`
- `*ip = 10;`

C++ adds the concept of constant pointer and pointer to a constant.

```
char * const ptr1 = "GOOD";       // constant pointer
```

We can not modify the address that ptr1 is initialized to.

```
int const *ptr2 = &m;             // Pointer to a constant.
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

const char * const cp = “xyz”;

This statement declares cp as a constant pointer to the string which has been declared a constant. In this case neither the address assigned to the pointer cp nor the contents it points can be changed.

Pointers are extensively used in C++ for memory management and achieving polymorphism.

Symbolic Constants

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

- Using the qualifier const and
- Using a set of integer constants using enum keyword.

In both C and C++ any value declared as const can not be modified by the program in any way. 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. const allows us to create typed constant instead of having to use #define to create constants that have no type information.

As with long and short, if we use the const modifier alone it defaults to int. For example

const size = 10;

Means **const int size = 10**;

The named constants are just like variables except that their values cannot be changed

C++ requires a const to be initialized. ANSI C does not require an initialize; if none is| given, it initializes the const to 0.

The scoping of const values differs. A const in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. In ANSI C, const values are global ii nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as static. To give a const value an external linkage so that it can be referenced from another file, we must explicitly define it as an extern in C++. Example:]

extern const total = 100; Another method of naming integer constants is by enumeration as under;

```
enum {X.Y.Z};
```

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to:

- **const x = 0;**
- **const y = 1;**
- **const z = 2;**

We can also assign values to x, y, and z explicitly. Example: enum(x = 100, y = 50, z = 200);
Such values can be any integer values.

Type Compatibility

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines **int, short int.** and **long int** as three different types. They must be cast when their values are assigned to one another. Similarly, **unsigned char, char,** and **signed char** are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way **char** constants are stored. In C, they are stored as ints, and therefore,

```
sizeof ('x')
```

is equivalent to

```
sizeof (int)
```

in C. In C++, however, **char** is not promoted to the size of **int** and therefore

```
sizeof('x')
```

equals

```
sizeof(char)
```

Declaration of Variables

We know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be declared at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration. Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type.

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

The example below illustrates this point.

```
int main()
{
   float x;

   float sum = 0;

   for (i = 0; i <; i++)
   {
      cin >> x;
      sum =  sum + x;
   }

   float average;

   average = sum/ (i-1);

   cout  << average;

   return 0;
}
```

The only disadvantage of this style of declaration is that we cannot see all the variables used in scope; at a glance.

Dynamic Initialization of Variables

In C a variable must be initialized using a constant expression, and the C compiler would fix the initialization code at the time of compilation. C++, however, permits initialization of the variables at run time. This is referred to as ***dynamic initialization***. In C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements

…….

……..

```
int n = strlen(string);

…

float area = 3.14159 * rad * rad;
```

thus, both declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The following two statements in the example of the previous section

```
float average;     // declare where it is necessary

average = sum / i;
```

can be combined into a single statement:

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

Dynamic **initialization** is extensively used in object-oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

Reference Variables

C++ introduces a new kind of variable known as **the *reference*** variable. A reference variable provides an *alias* (alternative name) for a previously defined variable. For example, if we make the variable **sum** a reference to the variable **total**, then **sum** and **total** can be used interchangeably to represent that variable. A reference variable is created as follows:

```
data-type & reference-name = variable-name


float total = 100;

float & sum = total;
```

total is a float type variable that has already been declared; sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object int the memory. Now, the statements

```
cout << total;
```
and
```
cout « sum;
```

print the value 100. The statement **total = total + 10;** will change

the value of both **total** and **sum** to **110**.

Likewise, the assignment **sum = 0;** will change the value of both the variables to zero:

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it names. It is important to not that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol &. Here, & is not an address operator notation **float &** means reference to **float.** Other examples are:

```
int n[10]; // x is alias for n[10]

int & x = n[10]; // initialize reference to a literal

char & a = '\n';
```

The variable **x** is an alternative to the array element **n[10].** The variable **a** is initial the newline constant.

This creates a reference to the otherwise unknown location when newline constant \n is stored.

Operators in C + +

C++ *has* a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator << and the extraction operator >>. Other new operators are:

: :	**Scope resolution operator**
::*	**Pointer-to-member declarator**
->*	**Pointer-to-member operator**
.*	**Pointer-to-member operator**
Delete	**Memory release operator**
Endl	**Line feed operator**
New	**Memory allocation operator**
Setw	**Field width operator**

In addition, C++ also allows us to provide new definitions to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used.

This process is known as *operator overloading.*

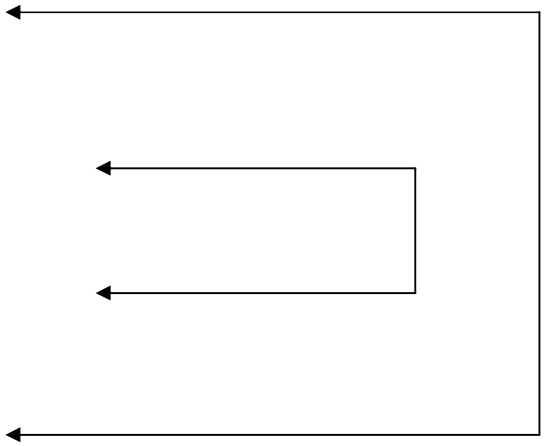
Scope Resolution Operator

Like C, C++ is also a block-structured language. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meaning in different blocks. The scope of the variable extends from the point of its declaration till the of the block containing the declaration. A variable declared inside a block is said to be local to that block. Consider the following segment of a program:

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

The two declarations of **x** refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable **x** declared in the first block, and vice versa.

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



Block2 is contained in block 1. Note that a ***declaration in an inner block hides a declaration of the same variable in an outer block*** and. 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, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator **::** called the ***scope resolution operator.*** This can be used to uncover a hidden variable. It takes the following form:

:: variable-name

This operator allows access to the global version of a variable. For example, **::count** means the global version of the variable **count** (and not the local variable **count** declared in that block). Following Program illustrates this feature.

```
#include <iostream>
using namespace std;
int m = 10;    // global m
int main()
{
   int m = 20;      // m redeclared, local to main
   {
      int k = m;     int m = 30;  // m declared again local to inner block
      cout «           "we are in inner block \n";
      cout «   "k = " « k « "\n";
      cout «   "m = " « m « "\n";
      cout «   "::m = " « ::m « "\n";
   }
      cout « "\nWe are in outer block \n";
      cout « "m = " « m «  "\n";
      cout « "::m = " « ::m « "\n";
    return 0;
}
```

The output of Program would be:

```
We are in  inner block k = 20
M = 30
::m = 10
We are  in outer block m =20 ::m  =  10
```

In the above program, the variable **m** is declared at three places, namely, outside the **main() function,** inside the **mainO function** and inside the **inner block.**

It is to be noted **::m** will always refer to the global **m**. In the inner block **::m** refers to the value **10** and not **20.** A major application of the scope resolution operator is in the classes to identify the ***class*** *to which* a member function belongs.

Member Dereferencing Operators

As you know, C++ permits us to define a class containing various types of data and function as members. C++ also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer-to-member operators. Following are these ***Member dereferencing operators***

Operator ***Function***

- ::* To declare a pointer to a member of a class
- * To access a member using object name and a pointer to that member
- —>* To access a member using a pointer to the object and a pointer to that member

Further details on these operators will be meaningful only after we discuss classes, therefore we defer the use of member dereferencing operators until then.

Memory Management Operators

C uses **mallocO** and **callocO** functions to allocate memory dynamically at run time. Similarly it uses the function **freeO** to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as free ***store*** operators.

An object can be created by using **new** and destroyed *by* using **delete,** as and when required. A data object created inside a block with **new**, will remain in existence until it is explicitly destroyed by using **delete.** Thus, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The new operator can he used to create objects of any type. It takes the following general form

Pointer-variable = new data-type

Here, ***pointer-variable*** is a pointer of type ***data-type***. The new operator allocates sufficient memory to hold a data object of type ***data-type*** and returns the address of the object. The ***data-type*** may be any valid data type. *The* ***pointer-variable*** holds the address of the memory space allocated. Examples:

p = new int;

q = new float;

where p is a pointer of type **int** and q is of type **float.** Here, p and q must have already been declared as pointers of' appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;

float *q = new float;
```

Subsequently, the statements

```
*p  =  25;
*q  = 7.5;
```

assign **25** to the newly created **int** object and **7.5** to the **float** object. We can also initialize the memory using the new operator. This is done as follows:

pointer-variable **=** **new** ***data-type(value):***

Here, value specifies the initial value. Examples

```
int *p = new int (25);
                                    float *q = new float(7.510);
```

As mentioned earlier, **new** can be used to create a memory space for any ***data-type*** including user-defined types such as **arrays, structures and classes.** The general form for a one-dimensional array is:

pointer-variable **=** ***new data-type[size];***

Here, size specifies the number of elements in the array. For example, the statement

```
int *p = new int [10];
```

creates a memory space for an array of 10 integers. p[0] will refer to the first element, p[1] to the second element, and so on.

When creating multi-dimensional arrays with new**,** all the array sizes must be supplied.

```
array_ptr =   new int[3] [5] [4];      //      legal
array_ptr =   new int[m][5][4];        //      legal
array_ptr =   new int[3][5][ ];        //      illegal
array_ptr =   new int[ ] [5][4];       //      Illegal
```

The first dimension may be a variable whose value is supplied at runtime. All other must be constants.

The application of **new** to class objects will be discussed.

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

delete pointer-variable;

*The **pointer-variable*** is the pointer that points to a data object created with **new**. Examples:

delete p;

delete q;

If we want to free a dynamically allocated array, we must use the following form of **delete:**

delete [size] pointer-variable;

The *size* specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example

delete [ ]p;

Will delete the entire array pointed to by p. What happens if sufficient memory is not available for allocation? In *such cases,* like **malloc( )** , **new** return a null pointer. Therefore, it may be a good idea to check for the pointer produced by **new** before using it. It is done as follows :

… …

… …

p = new int;

if (!p)

{

cout << "allocation failed \n";

}

… …

… …

The new operator offers following advantages over the function malloc

- **It automatically computes the size of the data object. We need not use the operator sizeof.**
- **It automatically returns the correct pointer type, so that there is no need to use a type cast**
- **It is possible to initialize the object while creating the memory space.**
- **Like any other operator, new and delete can be overloaded.**

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 the same effect as using the new line character "\n". For example, the statement

… …

```
cout  << "m = " << m << endl
      << "n = " << n << endl
      << "p = " << p << endl;
… …
```

would cause three line of output for each variable. If we assume the values of the variables as 2597, 14 and 175 respectively, the output will appear as follows.

```
m =  2597
n  =  14
p  =  175
```

It is important to note that this form is not the ideal output. It should rather appear as under:

```
m =  2597
n =    14
p =   175
```

Here, the numbers are ***right-justified***. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

```
cout « "sum = ";
cout « setw(5)  « sum « endl ;
```

The manipulator **setw(5)** specifies a field width **5** for printing the value of the variable sum. This value is right -justified within the field as shown below:

```
sum =  345
```

Program illustrates the use of endl and setw.

```
#include <iostream>
#iinclude <iomanip>     // for setw

using namespace std;
int main()
{
   int Basic = 950, Allowance = 95, Total = 1045;

   cout  «setw(10) « "Basic" «setw(10) « Basic « endl
   «    setw(10) « "Allowance"« setw(10) « Allowance « endl
   «    setw(10) « "Total" «setw(10) « Total « endl;

   return 0;

}
```

Output of this program is given below.

```
Basic          950

Allowance       95

Total         1045
```

Type Cast Operator

C++ permits explicit type conversion of variables or expressions using the type cast operator. Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. The following two versions are equivalent.

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

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

Examples :

```
average = sum / (float) I;    // C notation

average = sum / float(i);     // C++ notation
```

A **type-name** behaves as if it s a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example,

```
p = int * (q);
```

is illegal. In such cases, we must use C type notation.

```
p = (int *) q;
```

Alternatively, we can use **typedef** to create an identifier of the required type and use in the functional notation.

```
typedef int * int_pt;

p = int_pt(q) ;
```

Expressions and Their Types

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types

- **Constant expressions**
- **Integral expressions**
- **Float expressions**
- **Pointer expressions**
- **Relational expressions**
- **Logical expressions**
- **Bitwise expressions**

An expression may also use combinations of the above expressions. Such expressions are known as ***compound expressions***

Constant Expressions

Constant Expressions consist of only constant values. Examples:

15
20 + 5 / 2.0
'X'

Integral Expressions

Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

m
m * n - 5
m * 'x'
5 + int (2.0)

where m and n are integer variables.

Float Expressions

Float Expressions are those which, after all conversions, produce floating-point results. Examples:

x + y
x * y / 10
5 + float (lO)
10.75

where x and y are floating-point variables.

Pointer Expressions

Pointer Expressions produce address values. Examples:
&m
ptr
ptr + 1
"xyz"

where m is a variable and **ptr** is a pointer.

Relational Expressions

Relational Expressions yield results of type **bool** which takes a value **true** or **false.** Examples

x <= y
a + b = = c + d
m + n > 100

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known a.-***Boolean expressions.***

Logical Expressions

Logical Expressions combine two or more relational expressions and produces **bool** type results. Examples

```
a  > b && x == 10
x == 10  || y == 5
```

Bitwise Expressions

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples :

```
x << 3          // Shift three bit position to left
y>> 1           // Shift one bit position at right
```

Shift operators are often used for manipulation and division by power of two.

Special Assignment Expressions

Chained assignment

```
x = (y= 10);        Or
x = y = 10;
```

First 10 is assigned to y and then to x.

A chained statement can not be used to initialize variables at the time declaration. For instance, the statement

```
float a = b = 12.34;          // wrong
```

is illegal. This may be written as

```
float a = 12.34, b = 12.34;   // correct
```

Embedded assignment

```
x = (y = 50) + 10;
```

y= 50 is an assignment expression known as embedded assignment. Here. the value **50** is assigned to **y** and then the result **50 + 10 = 60** is assigned to **x**. This statement is identical to

```
y =  50;
x = y + 10;
```

Compound Assignment

Like C, C++ supports a ***compound assignment operator*** which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

x = x + 10;

may be written as

x += 10;

The operator **+=** is known as ***compound assignment operator*** or ***short-hand assignment operator***. The general form of the compound assignment operator is:

variable1 op= variable2;

where ***op*** is a **binary arithmetic operator**. This means that

variable1 = variable1 op variable2;

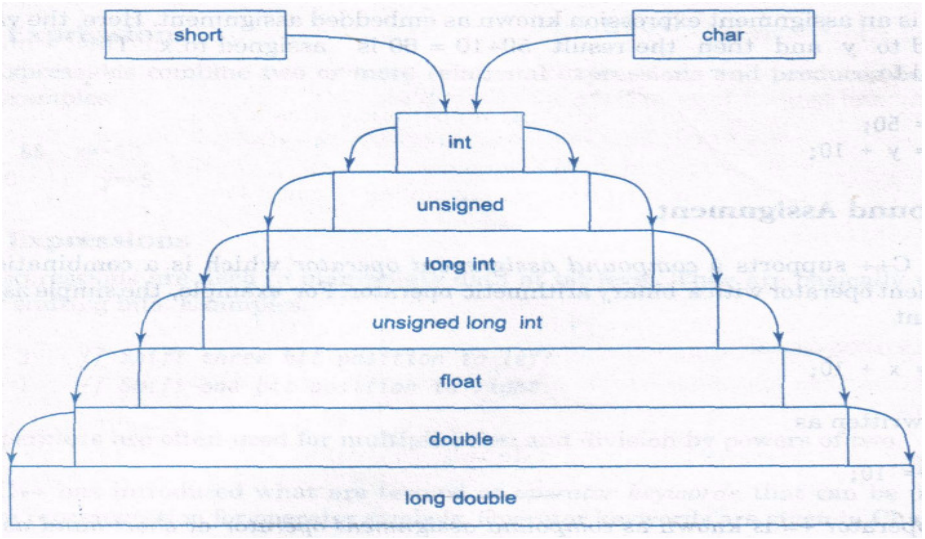
Implicit Conversions

We can mix data types in expressions. For example,

m = 5+2.75;

is a valid statement. Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is know[7]n as ***implicit*** or ***automatic conversion.***

When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the **"smaller"** type is converted to the **"wider"** type. For example, if one of the operand is an **int** and the other is a **float,** then **int** is converted into a **float** because a **float** is wider than an **int.** The "water-fall" model shown in following figure illustrates this rule.



Whenever a **char** or **short int** appears in an expression, it is converted to an **int.** This is called ***integral widening conversion.*** The implicit conversion is applied only after completing of all **integral widening conversions.**

Operator Overloading

Overloading means assigning different meanings to an operation, depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators. Actually, we have used the concept of overloading in C also. For example the operator * when applied to a pointer variable, gives the value pointed to by the pointer. But it is also commonly used for multiplying two numbers. The number and type of the operands decide the nature of operation to follow.

The input/output operators « and » are good examples of operator overloading. Although the built-in definition of the « operator is for shifting of bits, it is also used for displaying the values of various data types.

cout << 75.86

invokes the definition for displaying a **double** type value, and

cout << "Well Done

invokes the definition for displaying a **char** value.

Similarly we can define additional meanings to other C++ operators. For example, we can define + operator to add two structures or objects. Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators *(. and .*)*, conditional, scope resolution operator *(::)* and the size operator (**sizeof**).

Operator Precedence

Although C++ enables us to add multiple meanings to the operators, yet their association and precedence remain the same. For example, the multiplication operator will continue higher precedence than the add operator. Following Table gives the precedence and associatively of all the C++ operators. The groups are listed in the order of decreasing precedence. The labels *prefix and* ***postfix*** distinguish the uses of **++** and **- -** .Also, the symbols **+, -, *,** and **&** are used as both **unary** and **binary** operators.

Operator	**Associativity**
::	Left to Right
> . ( ) [ ] postfix ++ postfix – prefix ++ prefix - unary * unary & (type) sizeof new delet	Right to Left
-> * *	Right to Left
* / %	Left to Right
+ -	Left to Right
<< >>	Left to Right
== !=	Left to Right
&	Left to Right
^	Left to Right
|	Left to Right
&&	Left to Right
||	Left to Right
?:	Right to Left
= *= /= %= += = <<= >>= &= ^= |=	Right to Left
, (comma	Left to Right

Control Structures

In C++, a large number of functions are used that pass messages, and process the data contained in objects. A function is set up to perform a task. When the task is complex, many different algorithms can be designed to achieve the same goal. Some are simple comprehend, while others are not. Experience has also shown that the number of bugs that occur is related to the format of the program. The format should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later. One method of achieving the objective of an accurate, error-resistant and maintainable code is to use one or combination of the following three control structures:

1. **Sequence structure (straight line)**
2. **Selection structure (branching)**
3. **Loop structure (iteration or repetition)**

Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in following figure. This shows that C++ combines the power of structured programming with the object-oriented paradigm.

The If Statement

if statement is implemented in two forms

- **Simple if statement**
- **if...else statement**

C++ statements to implement in two forms:

Examples :

Form 1

```
if (expression is true)
{
    action1;
}
action2;
action3;
```

Form 2

```
if (expression is true)
{
    action 1;
}
else
{
    action2;
}
action 3;
```

switch statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
switch(expression)
 {
        casel:
        {
         actionl;
        }
         case2:
        {
         action2;
        }
        case3:
        {
         action3;
        }
        default:
        {
         action5;
        }
}
```

The do-while statement

The do-while is an ***exit-controlled*** loop. Based on a condition, the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
   actionl ;
 }
while(condition   is  true);

action2;
```

The while statement

This is also a loop structure, but is an ***entry-controlled*** one. The syntax is as follows:

```
while(condition  is  true)
{
   actionl;
 }
action2;
```

The for statement

The for is an ***entry -controlled*** loop and is used when an action is to be repeated **for a** predetermined number of times. The syntax is as follows:

```
for(initial value; test ; increment)
{
   actionl;
 }

action2;
```