1 Tokens

The C++ Tokens are the smallest individual units of a program.

C++ is the superset of C and so most constructs of C are legal in C++ with their meaning and usage unchanged. So tokens, expressions, and data types are similar to that of C.

The following tokens are available in C++ :

- Keywords
- Identifiers
- Constants
- Variables
- Operators

1.1 Keywords

Keywords are reserved words which have fixed meaning, and its meaning cannot be changed.

In the first group we put those that were also present in the C programming language and have been carried over into C++. There are 32 of these, and here they are:

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

There are another 30 reserved words that were not in C, are therefore new to C++, and here they are:

> *asm dynamic_cast namespace reinterpret_cast try*
>
> *bool explicit new static_cast typeid*
>
> *catch false operator template typename*
>
> *class friend private this using*
>
> *const_cast inline public throw virtual*
>
> *delete mutable protected true wchar_t*

1.2 Identifiers

Identifiers refers to the name of variables, functions, arrays, classes, etc. created by the user. Also, identifier names should have to be unique because these entities are used in the execution of the program.

The following rules are common to both c and c++ :

- Only alphabetic characters, digits and underscores are permitted.
- First letter must be an alphabet or underscore (_).
- The name cannot start with a digit.
- Identifiers are case sensitive(myVar and myvar are different variables).
- Reserved keywords can not be used as an identifier's name.

Some of the valid identifiers are: **shyam, _max, j_47, name10**.

And invalid identifiers are :**4xyz, x-ray, abc 2**.

1.3 Constants

Constants refers to fixed values assigned to the variables that do not change during the execution of a program.

There are two different ways to define constants in C++. These are:

- By using const keyword
- By using #define preprocessor

Syntax to declare a constant :

```
const [data_type] [constant_name]=[value];
```

The several kinds of c++ constants are:

- Integer Constants:

  Example: const int data = 5;

- Float Constants:

  Example: const float e = 2.71;

- Character Constants:

  Example: const char answer = ‘y’;

- String Constants

  Example: const char title[] = ‘‘C++ Program’’;

Example

```
#include <iostream.h>
int main()
{
        const int  max_length=100; //        integer constant
        const char choice='Y';               //        character constant
        const char title[]="c++ programming";        //        string constant
        const float temp=12.34;              //  float constant
        cout<<"max_length :"<<max_length<<endl;
        cout<<"choice :"<<choice<<endl;
        cout<<"title :"<<title<<endl;
        cout<<"temp :"<<temp<<endl;
        return 0;
}
```

1.4 Variable

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

Syntax to declare a variable

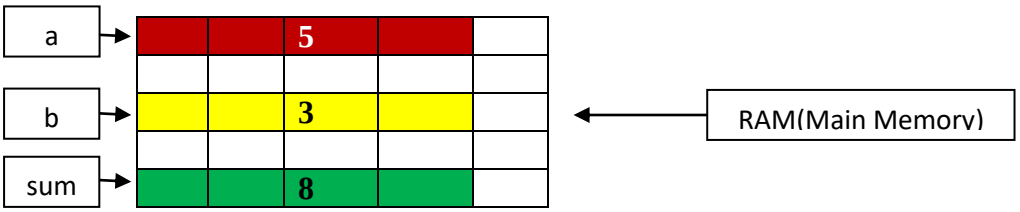
```
[data_type] [variable_name];
```

Variables are declared in two different places:

- Local variables: Inside Function
- Global variables: Outside of all function

Example

```
#include <iostream.h>

 int a,b;  // a and b are global variable

int main() {

  cout<<" Enter first number :";

  cin>>a;

  cout<<" Enter the second number:";

  cin>>b;

  int sum; //local variable

  sum=a+b;

  cout<<" Sum is : "<<sum <<"\n";

  return 0;

}
```



1.5 Operator

C++ operator is a symbol that is used to perform mathematical and logical operations. Operators and operands together form an experssion.

Example: a=b+c

a, b, and c: Operands

+, =: Operators

The several kinds of c++ operators are:

- Arithmetic Operators
- Increment and Decrement Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic Operators (Binary Operators)

Assume variable A holds 10 and variable B holds 20, then −

Operator	Description	Example
+	Addition	A + B will give 30
-	Subtraction	A - B will give -10
*	Multiplication	A * B will give 200
/	Division	B / A will give 2
%	Modulus	B % A will give 0

Increment and Decrement Operators (Unary Operators)

Operator	Description	Example
++	Increment	A++ will give 11
––	Decrement	A-- will give 9

Relational Operators

Operator	Description	Example
==	Is equal to	(A == B) is not true.
!=	Is not equal to	(A != B) is true.
>	Greater than	(A > B) is not true.
<	Less	(A < B) is true.
>=	Greater than or equal to	(A >= B) is not true.
<=	Less than or equal to	(A <= B) is true.

Logical Operators

Assume variable A holds 1 and variable B holds 0, then −

Operator	Description	Example
&&	And operator. Performs logical conjunction of two expressions.(if both expressions evaluate to True, result is True. If either expression evaluates to False, the result is False)	(A && B) is false.
||	Or operator. Performs a logical disjunction on two expressions.(if either or both expressions evaluate to True, the result is True)	(A || B) is true.
!	Not operator. Performs logical negation on an expression.	!(A && B) is true.

Bitwise Operators:

Assume variable A holds 6 and variable B holds 3, then –

A=0110

B=0011

A<<1 =1100

A>>1=0011

~A=1001

A&B=0010

A|B=0111

A^B=0101

Operator	Description	Example
<<	Binary Left Shift Operator	A<<1 will give 12 which is 1100
>>	Binary Right Shift Operator	A>>1 will give 3 which is 0011
~	Binary One's Complement Operator	~A will give 9 which is 1001
&	Binary AND Operator	A&B will give 2 which is 0010
^	Binary XOR Operator	A^B will give 5 which is 0101
|	Binary OR Operator	A|B will give 7 which is 0111

Assignment Operators

Operator	Description	Example
=	Assign	C = A + B will assign value of A + B into C
+=	Increments, then assign	C += A is equivalent to C = C + A
-=	Decrements, then assign	C -= A is equivalent to C = C - A
*=	Multiplies, then assign	C *= A is equivalent to C = C * A
/=	Divides, then assign	C /= A is equivalent to C = C / A
%=	Modulus, then assigns	C %= A is equivalent to C = C % A
<<=	Left shift and assigns	C <<= 2 is same as C = C << 2
>>=	Right shift and assigns	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assigns	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assigns	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assigns	C |= 2 is same as C = C | 2

Conditional or Ternary Operator (?:) in C++

The conditional operator is kind of similar to the if-else statement as it does follow the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.

Syntax:

```
variable = (Conditional Expression1) ? TRUE : FALSE
```

It is also called **ternary operators, as it** takes three operands to work .

Example:

```
#include <iostream.h>
int main () {
   // Local variable declaration:
   int x, y = 10;
   x = (y < 10) ? 30 : 40;
   cout << "value of x: " << x << endl;
   return 0;
}
```

Operator Precedence and Associativity:

Operator Precedence: Operator Precedence is the characteristics of operators that determine the evaluation order of operators in absence of brackets.

Note: Precedence from top to bottom is in descending order.

Category	Operator	Associativity
High Precedence(postfix)	() []	Left to right
Unary	+ - ! ~ ++ - - sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

For example: a+b*c

In the above example the precedence of * is higher than the +, so first * is solved then + is solved.

Associativity: Associativity is only used to determine which operator is evaluated first when two operator of same precedence are appeared together in expression.

For example: a-b+c

In the above example the precedence of both + and – is same, so now they are solved by using their associativity, the associativity of both – and + is from left to right, so first – is solved than the + is solved.

Example:

int a=10+2/3*4;

According to the highest precedence: /, *, +

First is solved / expression 10+0*4

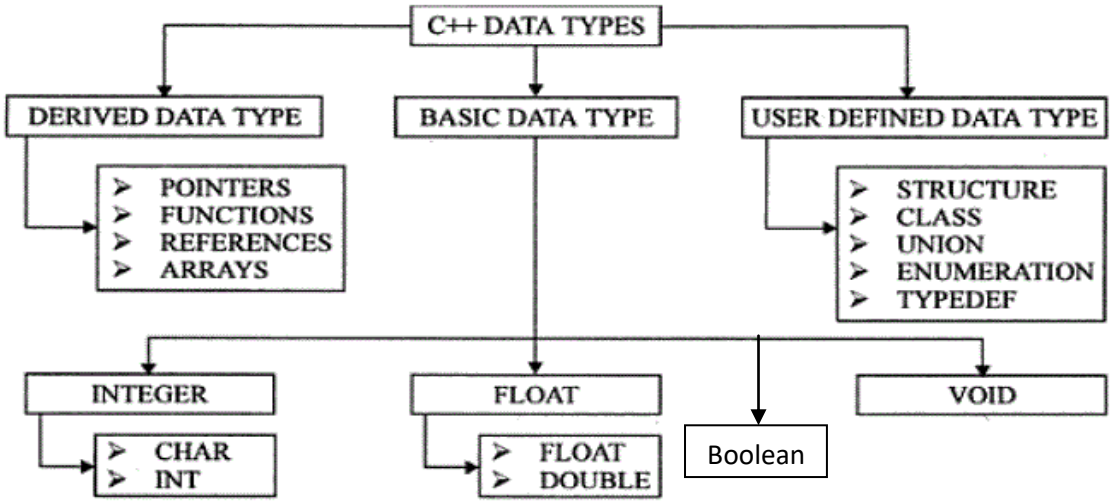
Then * is solved, so 10+0

Then + is solved 10

Then assignment is done, so a will store 10.

Data types in C++

- Basic data types
- Derived data types
- User defined data type



C++ data types

Basic data types

Data Types	Memory Size	Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127

int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-2,147,483,648 to 2,147,483,647
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 4,294,967,295
float	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	12 bytes	+/- 1.7e +/- 308 (~15 digits)

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

To find the size of variable, sizeof operator is used.

```
sizeof(dataType);
```

Boolean data type:

- Boolean data type is used for storing boolean or logical values. A boolean variable can store either **true or false**. Keyword used for boolean data type is **bool**.

Void data type:

- Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.

```
void display()
{
Cout<<"Hello";
}
```

Derived data type

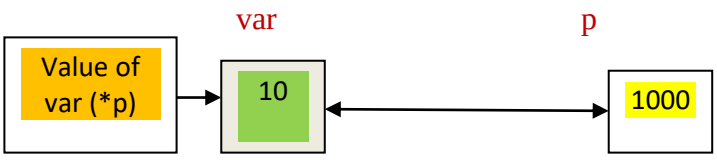
a) Pointer

- Pointer is a variable in C++ that holds the address of another variable.
- It is always denoted by '*' operator.
- To assign the address of variable to pointer we use **ampersand symbol** (&).

```
int *p;
int var;
p = &var;
```

Example:

```
#include <iostream.h>
#include<conio.h>
void main()
{
   int val = 10;
   int* p;
   p = &val;
   cout<<"p = "<<p<<"\n"; //address of val
   cout<<"val = "<<val<<"\n";
   cout<<"*p = "<<*p<<"\n";
    getch();
}
```



address=1000

b) Function

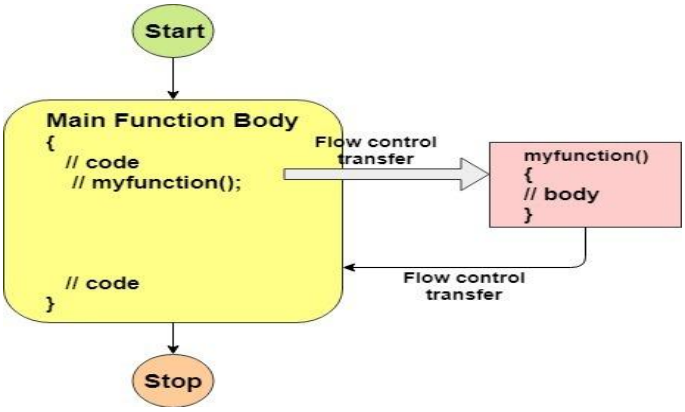
- A function is a block of code or program-segment that is defined to perform a specific well-defined task when call.

Syntax

```
Function_Return_Type  Function_Name (parameters)
{
//Function body
}
```

Each function has:

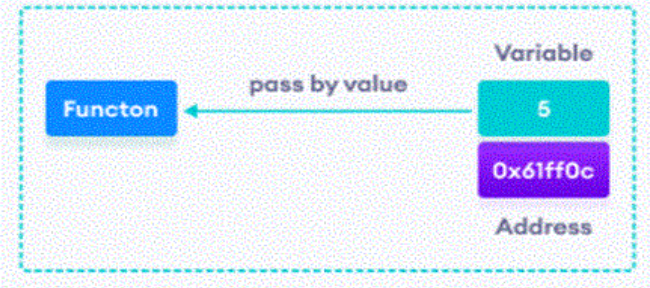
- Return type
- FunctionName
- Parameter List
- Function body



In C++, we have certain ways to pass parameters are:

Pass by Value

- In pass by value technique of parameter passing, the copies of values of actual parameters are passed to the formal parameters.
- The actual and formal parameters are stored at different memory locations. Thus, changes made to formal parameters inside the function do not reflect outside the function.



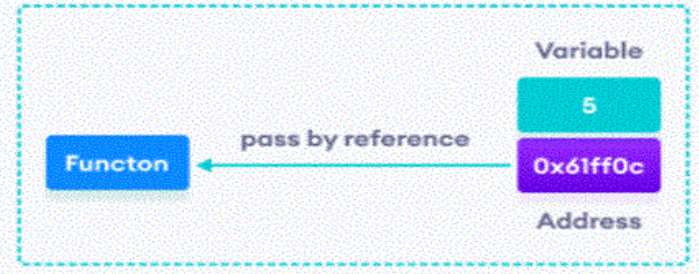
Example:

```
#include <iostream.h>
int sum(int a, int b) // function definition
{    int sum=0;
     sum = a+b;
     return sum;
}
int main()
{
 int x = 20, y = 40;
 int result = sum(x, y); // Calling the function
 cout <<"Result of addition:\n"<< result;
 return 0;
}
```

pass by reference

- In pass by reference technique of parameter passing, we use the references of actual parameters into formal parameters.

- The actual and formal parameters refer to the same memory locations. The changes made to formal parameters inside the function are actually reflected in actual parameters of the calling function.



Example:

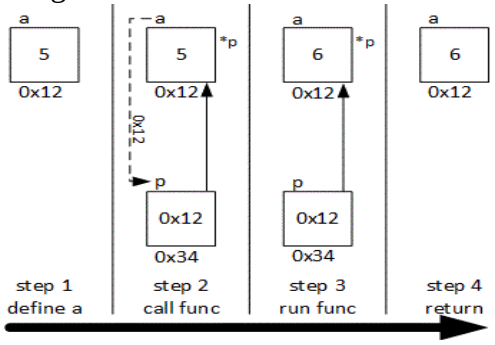
```
#include <iostream.h>
int sum(int &a, int &b) // function definition
{    int sum=0;
     sum = a+b;
     return sum;
}
int main()
{
 int x = 20, y = 40;
 int result = sum(x, y); // Calling the function
 cout <<"Result of addition:\n"<< result;
 return 0;
}
```

Pass by pointer

- In C++, we can also pass parameters to function using pointer variables.

- In pass by pointer technique of parameter passing, we use the address of actual parameters into formal parameters.

- The actual and formal parameters refer to the same memory locations. The changes made to formal parameters inside the function are actually reflected in actual parameters of the calling function.



Example:

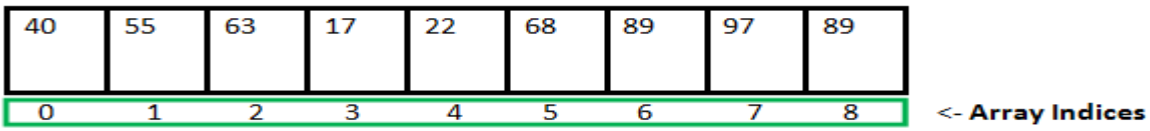
```
#include <iostream.h>
int sum(int *a, int *b) // function definition
{    int sum=0;
     sum = *a + *b;
     return sum;
}
int main()
{
 int x = 20, y = 40;
 int result = sum(&x, &y); // Calling the function
 cout <<"Result of addition:\n"<< result;
 return 0;
}
```

c) Array

• Array is a collection of elements of homogenous data type.

• C++ provides a **array** data structure, which stores a fixed-size sequential collection of elements of the same data type.

• All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Array Length = 9
First Index = 0
Last Index = 8

Syntax

```
Data_type arrayName [ arraySize ];
```

Example

Write a program to print the Addition of array element.

```
#include <iostream.h>
int main()
{
   // Array Derived Type
   int arr[4],i;
   int sum=0;
   cout<<"Enter the elements..\n";
   for(i =0; i<4; i++)
   {
      cin>>arr[i];
   }
  for(i=0; i<4; i++)
  {
```

```
   sum+=arr[i];
  }

  cout<<"Addition of elements of array:\n ";
  cout<<sum;
   return 0;
}
```

d) Reference

- It is used to define a reference variable.
- A reference is an alias, or an alternate name to an existing variable. For example, suppose you make peter a reference (alias) to paul, you can refer to the person as either peter or paul.
- A variable can be declared as a reference by putting '&' in the declaration.

syntax :

```
data type & reference_variable_name=variable name;
```

```
int qty=10;
int &qt=qty;
```

Example:

```
#include<iostream.h>
int main()
{
int x = 10;

// ref is a reference to x.
int & ref = x;

// Value of x is now changed to 20
ref = 20;
cout << "x = " << x << endl ;

// Value of x is now changed to 30
x = 30;
cout << "ref = " << ref << endl ;

return 0;
}
```

User defined data type
a) Structure

- **Structures in C++** are user defined data types which are used to store group of items of non-similar data types into a single type.
- The **'struct'** keyword is used to create a structure.
- Structure members can be initialized using curly braces '{}'.
- Structure members are accessed using dot (.) operator.

Syntax:

```
struct structureName
{
member1;
member2;
};
```

Example

```
#include <iostream.h>
struct Person
{
    char name[50];
    int age;
    float salary;
};

void main()
{
    Person p1;
    cout << "Enter Full name: ";
    cin.get(p1.name, 50);
    cout << "Enter age: ";
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;
    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p1.name << endl;
    cout <<"Age: " << p1.age << endl;
    cout << "Salary: " << p1.salary;
}
```

b) Classes

```
class student
{
char nm[30];
char sex[10];
int roll;
void input(void);
};
```

c) Union

- Union is a user-defined data type. All the members of union share same memory location. Size of union is decided by the size of largest member of union. If you want to use same memory location for two or more members, union is the best for that.
- The **'union'** keyword is used to create a structure.

Syntax

```
union union_name
{
  member definition;
};
```

d) Enumerated data type

- Enumeration (or enum) is a user defined data type in C++. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.
- The '**enum**' keyword is used to create a structure.

Syntax:

```
enum State {Working = 1, Failed = 0};
```

Example:

```
#include <iostream.h>
enum week {          Mon,
                     Tue,
                     Wed,
                     Thur,
                     Fri,
                     Sat,
                     Sun };
void main()
{
        enum week day;
        day = Wed;
        cout << day;
}
```

e) Typedef :

C++ allows you to define explicitly new data type names by using the keyword **typedef.** Using typedef does not actually create a new data class, rather it defines a name for an existing type.

Syntax:

```
typedef type_name;
```

Example:

```
#include <iostream.h>
typedef unsigned char Name;
int main()
{
        Name b1, b2;
        b1 = 'c';
        cout << " " << b1;
        return 0;
}
```

Type compatibility

- Type Compatibility between types refers to the similarity of two types to each other. Type compatibility is important during type conversions and operations.
- Type compatibility is very close to automatic or implicit type conversion.

The type compatibility is categorized into following three types by the compiler:

1 . Assignment compatibility

For example

```
float n1=12.5;
int n2=n1;
```

This assigning of float value to int type will result in loss of decimal value of n1. However, this type compatibility will not show any error but it give a warning “possible loss of data”.

2. Expression compatibility

For example

```
int n=3/2;
cout<<n;
```

Here the result will be 1. The actual result of 3/2 is 1.5 but because of incompatibility there will be loss of decimal value.

3. Parameter compatibility

Due to incompatibility in type of actual parameter and formal parameters loss of data occurs.

For example

```
void show(int n)
{
   cout<<”n=”<<n;
```

```
}
void main()
{
   show(8.2);
}
```
Here output will be n=8 due to incompatibility in actual and formal parameter type.

Type Conversion:

C++ allows us to convert data of one type to another type. This is called type conversion or type-casting.

C++ supports two types of Type Conversions:

1. **Implicit Type Conversion:**
   - Implicit type conversion Also known as automatic type conversion. There is no interference from the user in this type of conversion and the compiler directly carries out the conversion without any external force from the user.
   - When you are converting data from small sized data type to big sized data type is known as implicit casting.

     For example:

```
int num1 = 9;
double num2;
// implicit conversion, assigning int value to a double variable
num2 = num1;
```

2. **Explicit Type Conversion:**
   - Explicit type conversion is user-defined type-casting. The user casts or converts a value of one data type to another depending on the requirements.
   - When you are converting data from big sized data type to small sized data type is known as explicit casting.

     Syntax:

```
(data type) expression;
```

For example1:

```
float num1=5.8;
int num2;
// explicit conversion, assigning float value to a int variable
num2=(int) num1;
```

For example 2:

```
int num1=9;
// explicit conversion, assigning float value to a int variable
float num2= (float)num1/2;
```

Control Structures: Control structures are used to affect how statements are executed.

Three control Structures are:

1. Sequence structure (Straight line)
2. Selection structure (Branching)
3. Loop structure (Iteration or repetition)

1. Sequence structure:

- Statements are executed one after the other.
- Sequence structure is C++'s default behavior!

```
#include <iostream>
using namespace std;

int main(){//Global main function.
  //This main function executes a sequence of
  // statements and then terminates.  The
  // statements in the sequence do not
  // include decisions or loops.
  cout << "Hello Viet Nam\n";
  cout << "Hello Iraq\n";
  cout << "Hello America\n";
  cout << "Hello World\n";
  return 0;
}//end main function
```

2. Selection structure

- Used to choose among alternative courses of action.
- Making decisions

Three selection structures are:

i. If statement:

C++ if the condition is evaluated by the argument. If the condition is valid, it is executed.

Syntax

```
if (condition)
{
//code should be executed;
}
```

Example:

```
#include <iostream.h>
int main () {
int number = 10;
if (number % 2 == 0)
{
cout << "The Number you have Enter it is Even";
}
return 0;
}
```

ii. If else statement:

The statement C++ if-else also checks the condition. The declaration executes if the condition is true otherwise the block is carried out.

Syntax

```
if(condition)
{
//code should be executed;
}
else
{
//code should be executed;
}
```

Example

```
#include<iostream.h>
int main () {
int number = 15;
if (number % 2 == 0)
{
cout << "The Number you have Enter it is Even";
}
else
{
cout << "The Number you have Enter it is Odd";
}
return 0;
}
```

iii. Switch Statement

Switch case is a built from the multiple branch selection statement. C++ Switch statement executes a single statement.

Syntax

```
Switch(variable _name)
{
case value1:
//code should be executed;
break;
case value2:
//code should be executed;
break;
…
default:
//Code to execute if not all cases matched
break;
}
```

Question: Write a program to perform various mathematical operations of your choice.

3. Loop structure

- Used to repeat a set of instructions.
- Looping

Three loop structures are:

i. **For Loop Statement**: The 'for' loop used only when the number of iterations is known.

Syntax:

```
for(initialization; condition; incr/decr)
{
//code should be executed;
}
```

Example:

```
#include <iostream.h>
int main() {
for(int i = 2; i <= 20; i++)
{
cout << i << "\n";
}
}
```

ii. **While Loop Statement:** The 'while' loop is used when the number of iterations is not known.

Syntax:

```
while(condition)
{
//code should be executed;
incr/decr;
}
```

Example

```
#include<iostream.h>
int main() {
int i = 5;
while(i <= 20)
{
cout << i  << "\n";
i++;
}
}
```

iii. **Do while loop statement**: The 'do while' loop is used when the number of iterations is not known and the loop must be performed at least once.

Syntax:

```
do
{
//code should be executed;
}
while(condition);
```

Example

```
#include<iostream.h>
int main() {
int j = 2;
do{
cout << j << "\n";
j++;
} while (j <= 10) ;
}
```

iv. Break Statement: The 'break' statement is used for loop breakage or statement switching. The 'break' statement can also be used to jump out of a loop.

Syntax:

```
Jump-statement;
break;
```

Example

```
#include<iostream.h>
int main() {
for (int j = 1; j <= 10; j++)
{
if (j == 10)
{
break;
}
cout << j << "\n";
}
}
```

v. Continue Statement: The 'continue' is used for the continuation of the loop. The 'continue' statement breaks one iteration in the loop, if a specified condition occurs and continues with the next iteration in the loop.

Syntax:

```
 Jump-statement;
Continue;
```

Example

```
#include<iostream>
using namespace std;
int main()
{
for(int j = 1; j <= 10; j++){
if(j == 10){
continue;
}
cout << j << "\n";
}
}
```

scope resolution operator(::)

- The :: (scope resolution) operator is used to get hidden names due to variable scopes so that you can still use them. The scope resolution operator can be used as both unary and binary operators.

The scope resolution operator (::) is used for following purposes:

1) **To access a global variable when there is a local variable with same name:**

```
#include<iostream.h>
int x; // Global x
void main()
{
int x = 10; // Local x
cout << "Value of global x is " << ::x;
cout << "\nValue of local x is " << x;
}
```

2) **To define a function outside the class:**

```
#include<iostream.h>
class A
{
 public:
  // Only declaration
  void fun();
 };

// Definition outside class using ::
void A::fun()
{
cout << "fun() called";
}
void main()
{
A a;
a.fun();
}
```

3) **To access the static variables of class**:

```
#include <iostream.h>

class X {

  public:

  static int count;

};

int X::count = 10;  // define static data member
```

```
void main () {
  int X = 0;   // hides class type X
  cout << X::count << endl;   // use static member of class X
}
```

4) **In case of Inheritance:**

```
#include<iostream.h>
class A
{
protected:
        int x=10;
};
class B: public A
{
protected:
        int x=20;
public:
void fun()
{
        cout << "A's x is " << A::x;
        cout << "\nB's x is " << x;
}
};

void main()
{
        B b;
        b.fun();
}
```