Basics:

Variable:

- Variables are used to store information in the memory location.
- This means when you create a variable you reserve some space in memory.
- To store information in a variable we have various data types

  -character
  -integer
  -floating point
  -double floating point
  -boolean

Note: Based on the data type of a variable, the operating system allocates memory

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Type Modifiers:

- Signed
- Unsigned
- Short
- Long

Type	Typical Bit
char	1 byte
unsigned char	1byte
signed char	1 byte
int	4 bytes
unsigned int	4 bytes

signed int	4 bytes
short int	2 bytes
unsigned short int	2 bytes
signed short int	2 bytes
long int	4 bytes
signed long int	8 bytes
unsigned long int	4 bytes
long long int	8 bytes
unsigned long long int	8 bytes
float	4 bytes
double	8 bytes
long double	12 bytes
wchar_t	2 or 4 bytes

You can also find size of type using program:

Program

```
#include<iostream>
using namespace std
int main()
{
        cout<< " size of char:" <<sizeof(char)<<endl;
        cout<< " size of int:" <<sizeof(int)<<endl;
cout<< " size of float:" <<sizeof(float)<<endl;
        return 0;
}
```

Output

size of char: 1

size of int: 4

size of float: 4

endl - which insert new line

Example Program in C++

```
#include<iostream>
usingnamespace std;

// main() is where program execution begins.
int main(){
cout<<"Hello World";// prints Hello World
return0;
```

```
}
```

Output

Hello World

Local variable:

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known outside of the function.

Program

```
#include<iostream>
usingnamespace std;

int main (){
// Local variable declaration:
intnum1,num2;
intsum;

// actual initialization
num1=10;
num2=20;
sum=num1+num2;

cout<<sum;

return0;
}
```

Output

30

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

Program

```
#include<iostream>
usingnamespace std;

// Global variable declaration:
```

```
intsum;

int main (){
// Local variable declaration:
intnum1,num2;

// actual initialization
num1=10;
num2=20;
sum=num1+num2;

cout<<sum;

return0;
}
```

Output

30

***Note:** if the program has same name for local and global variables but value of local variable inside a function will take preference.

Program

```
#include<iostream>
usingnamespace std;

// Global variable declaration:
intnum=20;

int main (){
// Local variable declaration:
intnum=10;

cout<<num;

return0;
}
```

Output

10

Defining constants

In two ways, we can define a constant

- Using **#define** preprocessor
- Using **const** keyword

Using **#define** preprocessor can define a constant

syntax

```
#define identifier value
```

Program

```
#include<iostream>
usingnamespace std;

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'// this gives new line

int main(){
int area;

   area = LENGTH * WIDTH;
cout<< area;
cout<< NEWLINE;
return0;
}
```

output

50

Using **const** keyword

You can use **const** prefix to declare constants with a specific type

Syntax

```
const type variable = value;
```

Program

```
#include<iostream>
usingnamespace std;

int main(){
constint  LENGTH =10;
constint  WIDTH  =5;
constchar NEWLINE ='\n';
int area;

   area = LENGTH * WIDTH;
cout<< area;
cout<< NEWLINE;
return0;
}
```

Output

50

Operators:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic operator

Program

```
#include <iostream>
using namespace std;
int main()
{
int num1,num2,Add,Sub,Mul,Div,Mol;
cout<<"Enter a first number : ";
cin>>num1;
cout<<"\nEnter the second number\n";
cin>>num2;
 //Addition
 Add=num1+num2;
cout<< "\n Addition:"<<Add;
 //Subtraction
 Sub=num1-num2;
cout<<"\n Subtraction:"<<Sub;
 //Multiplication
Mul=num1*num2;
cout<<"\n Multiplication:"<<Mul;
  //Divide
Div=num1/num2;   //it returns the quotient
cout<<"\n Divide:"<<Div;
  //Modulus
 Mol=num1%num2;   //it returns the reminder
cout<<"\n Modulus:"<<Mol;
//Increment and Decrement
 num1++; //it increases by plus 1
 num2--; // it decreases by minus 1
cout<<"\n Increment:"<<num1;
cout<<"\n Decrement:"<<num2;
 return 0;
```

Output:

Enter the first number: 8

Enter the second number: 5

Addition: 13

Subtraction: 3

Multiplication: 40

Divide: 1

Modulus: 3

Increment: 9

Decrement:4

Relational operator

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.

<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Program

```
#include<iostream>
usingnamespace std;

main(){
int a =21;
int b =10;
int c ;

if( a == b ){
cout<<" a is equal to b"<<endl;
}else{
cout<<" a is not equal to b"<<endl;
}

if( a < b ){
cout<<"a is less than b"<<endl;
}else{
cout<<"a is not less than b"<<endl;
}

if( a > b ){
cout<<" a is greater than b"<<endl;
}else{
cout<<" a is not greater than b"<<endl;
}

/* Let's change the values of a and b */
   a =5;
   b =20;
if( a <= b ){
cout<<"a is either less than \ or equal to b"<<endl;
}

if( b >= a ){
cout<<" b is either greater than \ or equal to b"<<endl;
}

return0;
}
```

Output:

a is not equal to b
a is not less than b
a is greater than b
a is either less than or equalto b
b is either greater than or equal to b

Logical operator

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

Program

```
#include<iostream>
usingnamespace std;

main(){
int a =5;
int b =20;
int c ;

if(a && b){
cout<<" Condition is true"<<endl;
}

if(a || b){
cout<<" Condition is true"<<endl;
```

```
}

/* Let's change the values of  a and b */
   a =0;
   b =10;

if(a && b){
cout<<" Condition is true"<<endl;
}else{
cout<<" Condition is not true"<<endl;
}

if(!(a && b)){
cout<<" Condition is true"<<endl;
}

return0;
}
```

Output

Condition is true
Condition is true
Condition is not true
Condition is true

Bitwise operator

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Operator	Description	Example

&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Program

Assume if A = 60; and B = 13; now in binary format

```
#include<iostream>
usingnamespace std;

main(){
unsignedint a =60;  // 60 = 0011 1100
unsignedint b =13;  // 13 = 0000 1101
int c =0;

   c = a & b;// 12 = 0000 1100
cout<<" Value of c is : "<< c <<endl;

   c = a | b;// 61 = 0011 1101
cout<<" Value of c is: "<< c <<endl;

   c = a ^ b;// 49 = 0011 0001
cout<<" Value of c is: "<< c <<endl;

   c =~a;// -61 = 1100 0011
cout<<" Value of c is: "<< c <<endl;

   c = a <<2;// 240 = 1111 0000
```

```
cout<<" Value of c is: "<< c <<endl;

   c = a >>2;// 15 = 0000 1111
cout<<" Value of c is: "<< c <<endl;

return0;
}
```

Output

Value of c is:12
Value of c is: 61
Value of c is: 49
Value of c is: -61
Value of c is: 240
Value of c is: 15

Assignment Operator

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Program

```
#include<iostream>
usingnamespace std;

main(){
int a =21;
int c ;

   c =  a;
cout<<" =  Operator, Value of c = : "<<c<<endl;

   c +=  a;
cout<<" += Operator, Value of c = : "<<c<<endl;

   c -=  a;
cout<<" -= Operator, Value of c = : "<<c<<endl;

c*=  a;
cout<<" *= Operator, Value of c = : "<<c<<endl;

c/=  a;
cout<<" /= Operator, Value of c = : "<<c<<endl;

c  =200;
c%=  a;
cout<<" %= Operator, Value of c = : "<<c<<endl;

   c <<=2;
cout<<" <<= Operator, Value of c = : "<<c<<endl;

   c >>=2;
cout<<" >>= Operator, Value of c = : "<<c<<endl;

c&=2;
cout<<" &= Operator, Value of c = : "<<c<<endl;
```

```
c^=2;
cout<<" ^= Operator, Value of c = : "<<c<<endl;

c|=2;
cout<<" |= Operator, Value of c = : "<<c<<endl;

return0;
}
```

Output

```
=  Operator, Value of c = : 21
+= Operator, Value of c = : 42
-= Operator, Value of c = : 21
*= Operator, Value of c = : 441
/= Operator, Value of c = : 21
%= Operator, Value of c = : 11
<<= Operator, Value of c = : 44
>>= Operator, Value of c = : 11
&= Operator, Value of c = : 2
^= Operator, Value of c = : 0
|= Operator, Value of c = : 2
```

Misc Operator

Sl.No	Operator & Description
1	**sizeof** sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	**Condition ? X : Y** Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
3	**,** Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	**. (dot) and -> (arrow)** Member operators are used to reference individual members of classes, structures, and unions.
5	**Cast**

	Casting operators convert one data type to another. For example, int(2.2000) would return 2.
6	**&** Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
7	*** Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

Operators Precedence in C++

- Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −
- For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.
- Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* &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

cProgram

```
#include<iostream>
usingnamespace std;

main(){
int a =20;
int b =10;
int c =15;
int d =5;
int e;

   e =(a + b)* c / d;// ( 30 * 15 ) / 5
cout<<"Value of (a + b) * c / d is :"<< e <<endl;

   e =((a + b)* c)/ d;// (30 * 15 ) / 5
cout<<"Value of ((a + b) * c) / d is  :"<< e <<endl;

   e =(a + b)*(c / d);// (30) * (15/5)
cout<<"Value of (a + b) * (c / d) is  :"<< e <<endl;

   e = a +(b * c)/ d;//  20 + (150/5)
cout<<"Value of a + (b * c) / d is  :"<< e <<endl;
```

```
return0;
}
```

Output

Value of (a + b) * c / d is :90
Value of ((a + b) * c) / d is :90
Value of (a + b) * (c / d) is :90
Value of a + (b * c) / d is :50

Loops in C++

- when you need to execute a block of code several number of times. Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages.

Sl.No	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop Like a 'while' statement, except that it tests the condition at the end of the loop body.
4	nested loops You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

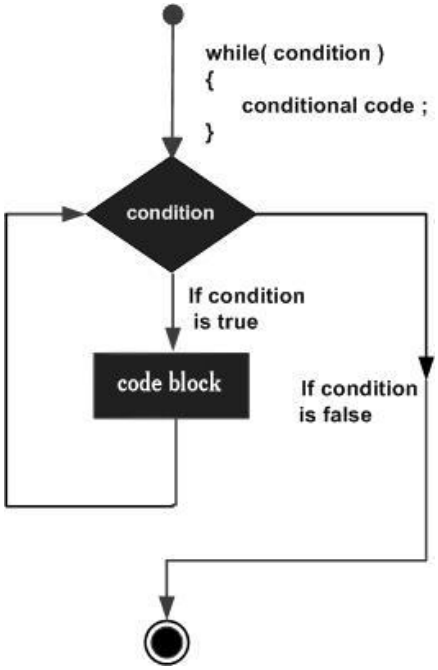
While loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax

```
while(condition) {
   statement(s);
}
```

Flow diagram



Program

```
#include<iostream>
usingnamespace std;

int main (){
// Local variable declaration:
int a =10;

// while loop execution
while( a <20){
cout<<"value of a: "<< a <<endl;
      a++;
}

return0;
}
```

Output

value of a: 10
value of a: 11
value of a: 12

value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

For loop

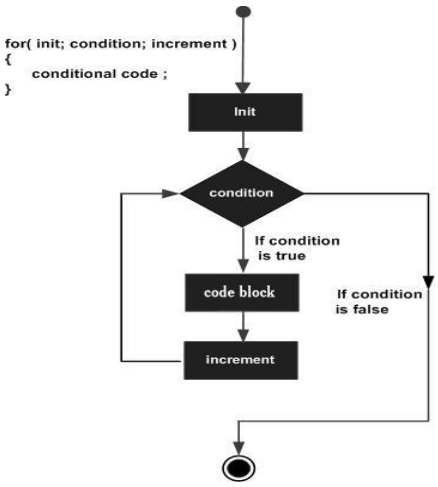
A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

```
for ( init; condition; increment ) {
   statement(s);
}
```

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram



Program

```
#include<iostream>
usingnamespace std;

int main (){
// for loop execution
for(int a =10; a <20; a = a +1){
cout<<"value of a: "<< a <<endl;
}

return0;
}
```

Output

value of a: 10

value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

do while loop

do...while loop checks its condition at the bottom of the loop.

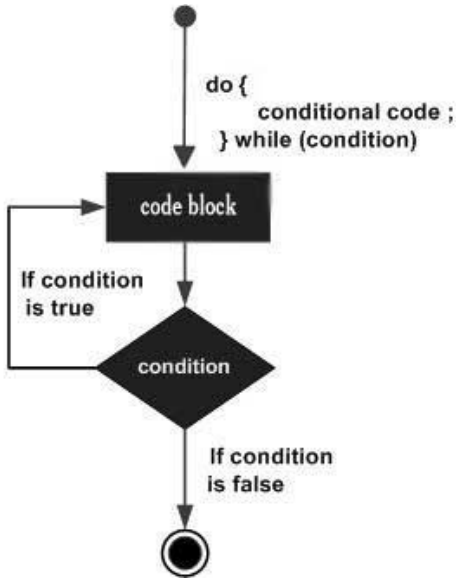
Syntax

```
do {
   statement(s);
}
while( condition );
```

The conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram



Program

```
#include<iostream>
usingnamespace std;

int main (){
// Local variable declaration:
int a =10;

// do loop execution
do{
cout<<"value of a: "<< a <<endl;
      a = a +1;
}while( a <20);

return0;
```

```
}
```

Output

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Nested for loop

Syntax

```
for ( init; condition; increment
) {
for ( init; condition; increment
) {
       statement(s);
    }
   statement(s); // you can put
more statements.
}
```

Program

```
#include<iostream>
usingnamespace std;

int main (){
inti, j;

for(i=2;i<100;i++){
for(j =2; j <=(i/j); j++)
if(!(i%j))break;// if factor found, not prime
if(j >(i/j))cout<<i<<" is prime\n";
}

return0;
}
```

Output

```
2 is prime
```

```
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

Loop control structure

Sl.No	Control Statement & Description
1	break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	goto statement Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

Decision making statements in C++

Sl.No	Statement & Description

1	if statement An ‘if’ statement consists of a boolean expression followed by one or more statements.
2	if...else statement An ‘if’ statement can be followed by an optional ‘else’ statement, which executes when the boolean expression is false.
3	switch statement A ‘switch’ statement allows a variable to be tested for equality against a list of values.
4	nested if statements You can use one ‘if’ or ‘else if’ statement inside another ‘if’ or ‘else if’ statement(s).
5	nested switch statements You can use one ‘switch’ statement inside another ‘switch’ statement(s).

if statement

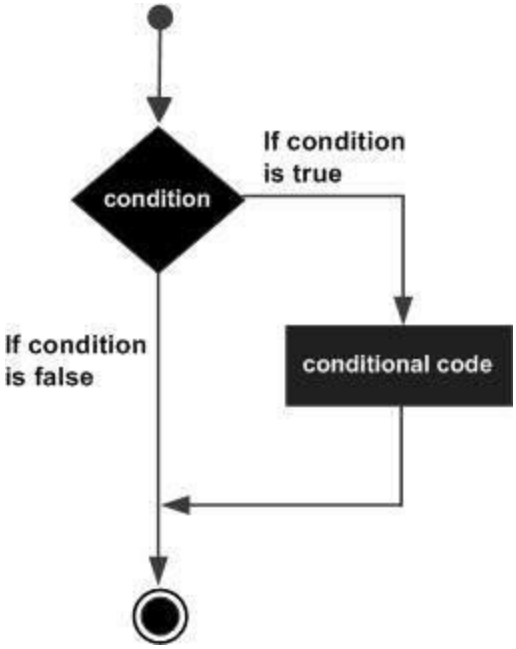
An **if** statement consists of a boolean expression followed by one or more statements.

Syntax

```
if(boolean_expression) {
   // statement(s) will execute if the
boolean expression is true
}
```

If the boolean expression evaluates to **true**, then the block of code inside the if statement will be executed. If boolean expression evaluates to **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Program

```
#include<iostream>
usingnamespace std;

int main (){
// local variable declaration:
int a =10;

// check the boolean condition
if( a <20){
// if condition is true then print the following
cout<<"a is less than 20"<<endl;
}
cout<<"value of a is : "<< a <<endl;

return0;
}
```

Output

a is less than 20
value of a is : 10

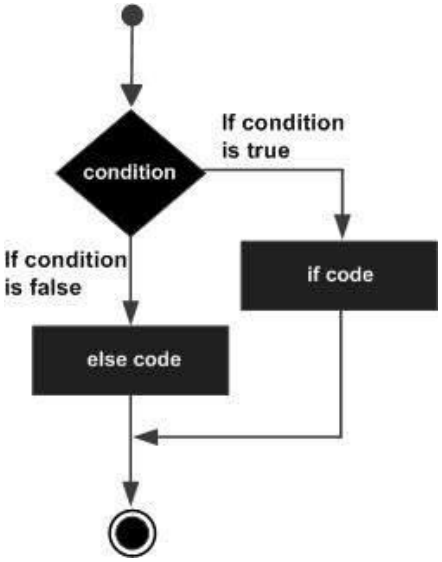
if else statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

Syntax

```
if(boolean_expression) {
   // statement(s) will execute if the boolean
expression is true
} else {
  // statement(s) will execute if the boolean
expression is false
}
```

Flow Diagram



Program

```
#include<iostream>
usingnamespace std;

int main (){
// local variable declaration:
int a =100;

// check the boolean condition
if( a <20){
// if condition is true then print the following
cout<<"a is less than 20;"<<endl;
}else{
// if condition is false then print the following
cout<<"a is not less than 20"<<endl;
}
cout<<"value of a is : "<< a <<endl;

return0;
}
```

Output

a is not less than 20
value of a is : 100

if else if statement

An **if** statement can be followed by an optional **else if...else** statement, which is very usefull to test various conditions using single if...else if statement.

Syntax

```
if(boolean_expression 1) {
   // Executes when the boolean
expression 1 is true
} else if(boolean_expression 2) {
   // Executes when the boolean
expression 2 is true
} else if(boolean_expression 3) {
   // Executes when the boolean
expression 3 is true
} else {
   // executes when the none of the
above condition is true.
}
```

Program

```
#include<iostream>
usingnamespace std;

int main (){
// local variable declaration:
int a =100;

// check the boolean condition
if( a ==10){
// if condition is true then print the following
cout<<"Value of a is 10"<<endl;
}elseif( a ==20){
// if else if condition is true
cout<<"Value of a is 20"<<endl;
}elseif( a ==30){
// if else if condition is true
cout<<"Value of a is 30"<<endl;
}else{
// if none of the conditions is true
cout<<"Value of a is not matching"<<endl;
}
cout<<"Exact value of a is : "<< a <<endl;

return0;
```

```
}
```

Output

Value of a is not matching
Exact value of a is : 100

Switch statement

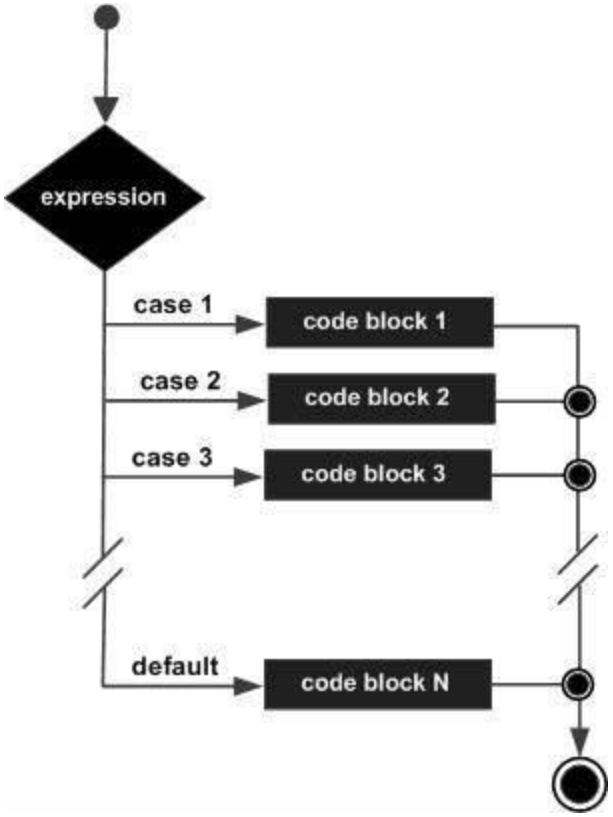
A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

```
switch(expression) {
case constant-expression  :
      statement(s);
      break; //optional
case constant-expression  :
      statement(s);
      break; //optional

   // you can have any number of case
statements.
default : //Optional
      statement(s);
}
```

- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram



Program

```
#include<iostream>
usingnamespace std;

int main (){
// local variable declaration:
char grade ='D';

switch(grade){
case'A':
cout<<"Excellent!"<<endl;
break;
case'B':
      cout<<"Good"<<endl;
break;
case'C':
cout<<"Well done"<<endl;
break;
case'D':
cout<<"You passed"<<endl;
break;
case'F':
cout<<"Better try again"<<endl;
break;
default:
cout<<"Invalid grade"<<endl;
}
cout<<"Your grade is "<< grade <<endl;
```

```
return0;
}
```

Output

You passed
Your grade is D

Contidional ? : operator

```
Exp1 ?Exp2 : Exp3;
```

```
if(y < 10) {
   var = 30;
} else {
   var = 40;
}
```

This can be written as

```
var = (y < 10) ? 30 : 40;
```

Program

```
#include<iostream>
usingnamespace std;

int main (){
// Local variable declaration:
int x, y =10;

   x =(y <10)?30:40;
cout<<"value of x: "<< x <<endl;

return0;
}
```

Output

```
value of x: 40
```

Unit I

Principles of Object- Oriented Programming – Beginning with C++ - Tokens, Expressions and Control Structures – Functions in C++

CHARACTERISTICS OF AN OBJECT ORIENTED PROGRAMMING LANGUAGE

OBJECT-ORIENTED PROGRAMMING

C++ fully supports object-oriented programming, including the four pillars of object-oriented development. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function

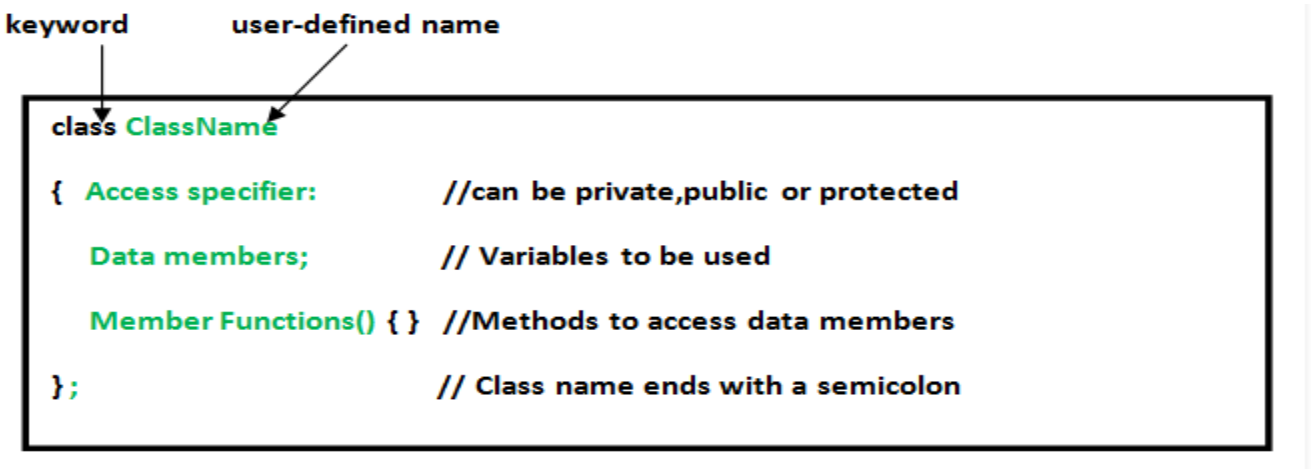
- Class
- Objects
- Encapsulation
- Data Abstraction
- Polymorphism
- Inheritance

- Dynamic Binding
- Message Passing

1.CLASS

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

- A Class is a user-defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- A class is defined in C++ using keyword class followed by the name of class.
- The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution **::** operator along with class name and function name.

2.OBJECT:

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Declaring Objects:

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax

```
ClassNameObjectName;
```

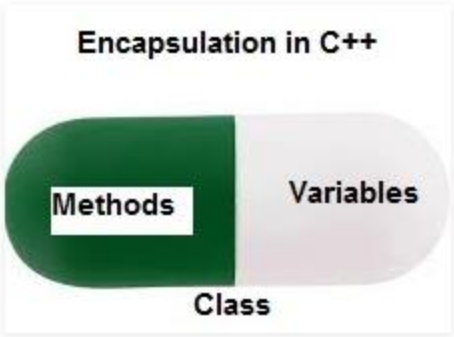
EXAMPLE

```
class person
{
char name[20];
int id;
public:
voidgetdetails(){}
};
int main()
{
person p1; // p1 is an object
}
```

3. ENCAPSULATION

Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. Example, the data of any of the section like sales, finance or accounts are hidden from any other section.



4.DATA ABSTRACTION

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

- **ABSTRACTION USING CLASSES:**
  We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

- **ABSTRACTION IN HEADER FILES**:
  One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power

  of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Advantages of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

5. POLYMORPHISM

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person possess different behaviour in different situations. This is called polymorphism.
- An operation may exhibit different behaviourin different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

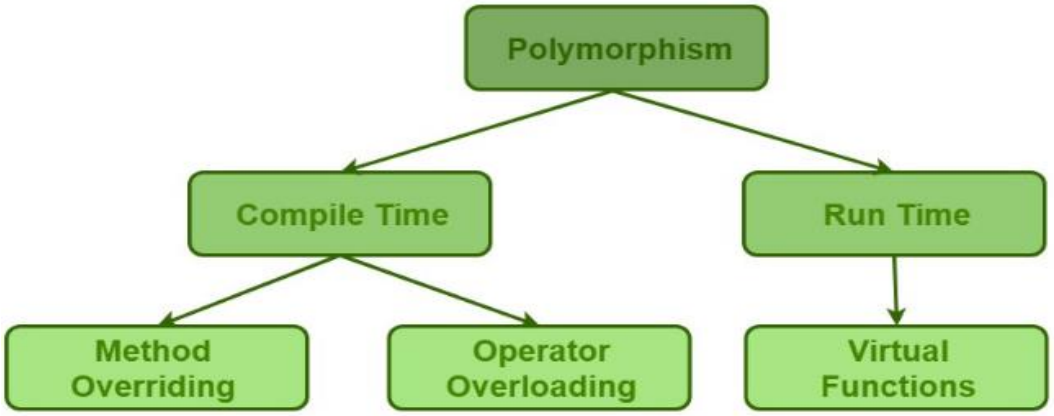
Operator Overloading:

- The process of making an operator to exhibit different behaviour in different instances is known as operator overloading.

Function Overloading:

- Function overloading is using a single function name to perform different types of tasks.

  Polymorphism is extensively used in implementing inheritance.



6. INHERITANCE

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**:

  The class that inherits properties from another class is called **Sub class or Derived Class**.
- **Super Class**:

  The class whose properties are inherited by sub class is called **Base Class or Super class.**
- **Reusability**:

  Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can

derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Implementing Inheritance InC++

For creating a sub-class which is inherited from the base class we have to follow the below syntax.

Syntax

```
classsubclass_name : access_modebase_class_name
{
  //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

filter_none

7. Dynamic Binding:

In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

8. Message Passing:

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Benefits of OOPs

- Through, inheritance, we can eliminate redundate code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of developing time and higher productivity.
- The principle of data hiding helps the programmer to build secure program that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

Object- oriented languages

Object-oriented programming concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially designed to support the OOP concepts makes it easier to implement them.

The language should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Objects-based programming languages
2. Object-oriented programming languages

Object-based Programming language is the style of programming that primarily supports encapsulation and object identity. Major feature that are required for object-based programming are;

1. Data encapsulation
2. Data hiding and access mechanisms
3. Automatic initialization and clear-up of objects
4. Operator overloading

Object-oriented programmingincorporates all the object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can thereof be characterized by following statement:

Object-based features + inheritance + dynamic binding

Data abstraction refers to putting together essential feature without including background details.

Inheritance is process by which objects of one class acquire properties of objects of another class

Polymorphism means one name, multiple forms. It allows us to have more than one function with the same name in a program. It also allows overloading of operators so that an operation can exhibit different behaviours in different instances.

Dynamic binding means that the code associated with a given procedure is not known until the time of the call at run-time

Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Object-oriented technology offers several benefits over the conventional programming methods—the most important one begin the reusability.

STANDARD LIBRARIES

Standard C++ consists of three important parts −

- The core language giving all the building blocks including variables, data types and literals, etc.
- The C++ Standard Library giving a rich set of functions manipulating files, strings, etc.
- The Standard Template Library (STL) giving a rich set of methods manipulating data structures, etc.

THE ANSI STANDARD

- The ANSI standard is an attempt to ensure that C++ is portable; that code you write for Microsoft's compiler will compile without errors, using a compiler on a Mac, UNIX, a Windows box, or an Alpha.
- The ANSI standard has been stable for a while, and all the major C++ compiler manufacturers support the ANSI standard.

BEGINNING WITH C++

- C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.
- C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.
- C++ was developed by BjarneStroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.
- C++ is a superset of C, and that virtually any legal C program is a legal C++ program.
- C++ is a bottom-up approach ie.code is developed for modules and then these modules are integrated with main() function.

Note − A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

USE OF C++

- C++ is used by hundreds of thousands of programmers in essentially every application domain.

- C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under realtime constraints.
- C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.
- Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.
- The most important thing while learning C++ is to focus on concepts.
- The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.

- C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency

APPLICATIONS OF C++ PROGRAMMING

As mentioned before, C++ is one of the most widely used programming languages. It has it's presence in almost every area of software development. I'm going to list few of them here:

- **Application Software Development** - C++ programming has been used in developing almost all the major Operating Systems like Windows, Mac OSX and Linux. Apart from the operating systems, the core part of many browsers like Mozilla Firefox and Chrome have been written using C++. C++ also has been used in developing the most popular database system called MySQL.
- **Programming Languages Development** - C++ has been used extensively in developing new programming languages like C#, Java, JavaScript, Perl, UNIX's C Shell, PHP and Python, and Verilog etc.
- **Computation Programming** - C++ is the best friends of scientists because of fast speed and computational efficiencies.
- **Games Development** - C++ is extremely fast which allows programmers to do procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines.
- **Embedded System** - C++ is being heavily used in developing Medical and Engineering Applications like software for MRI machines, high-end CAD/CAM systems etc.

This list goes on, there are various areas where software developers are happily using C++ to provide great software. I highly recommend you to learn C++ and contribute great software to the community.

STRUCTURE OF C++

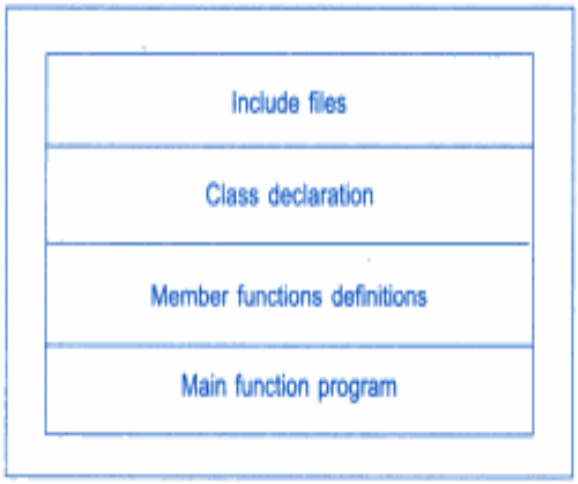


Fig. 2.3 ⇔ Structure of a C++ program

C++ TOKENS

Smallest individual units in a program are known as tokens. C++ has following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

C++ CONTROL STRUCTURES

- When a program runs, the code is read by the compiler line by line (from top to bottom, and for the most part left to right). This is known as "**code flow.**"
- When the code is being read from top to bottom, it may encounter a point where it **needs to make a decision**. Based on the decision, the program may jump to a different part of the code. It may even make the compiler re-run a specific piece again, or just skip a bunch of code.

C++ SYNTAX

- The syntax is a layout of words, expression, and symbols.
- Well, it's because an email address has its well-defined syntax. You need some combination of letters, numbers, potentially with underscores (_) or periods (.) in between, followed by an at the rate (@) symbol, followed by some website domain (company.com).
- So, syntax in a programming language is much the same. They are some well-defined set of rules that allow you to create some piece of well-functioning software.

- But, if you don't abide by the rules of a programming language or syntax, you'll get errors.

C++ TOOLS

- In the real world, a tool is something (usually a physical object) that helps you to get a certain job done promptly.
- Well, this holds true with the programming world too. A tool in programming is some piece of software which when used with the code allows you to program faster.
- There are probably tens of thousands, if not millions of different tools across all the programming languages.
- Most crucial tool, considered by many, is an IDE, an **Integrated Development Environment.** An IDE is a software which will make your coding life so much easier. IDEs ensure that your files and folders are organized and give you a nice and clean way to view them.

C++ VARIABLES

- Variables are the backbone of any programming language.
- A variable is merely a way to store some information for later use. We can retrieve this value or data by referring to a "word" that will describe this information.
- Once declared and defined they may be used many times within the scope in which they were declared

A variable provides us with a named storage capability. It allows programmer to manipulate data as per the need. Every variable in C++ has a type. The variable type helps to determine the size and layout of the variable's memory map, the range of values that can be stored within that memory, and the set of operations that can be applied to it.

Variable Name or Identifiers

Identifiers can be composed of some letters, digits, and the underscore character or some combination of them. No limit is imposed on name length.

Rules for Identifiers

- begin with either a letter or an underscore ('_').
- And are case-sensitive; upper and lowercase letters are distinct.

FUNCTIONS IN C++

- A function is a group of statements that together perform a task.
- Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

Create a Function

C++ provides some pre-defined functions, such as main(), which is used to execute code. But you can also create your own functions to perform certain actions.

To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses **()**:

Syntax

```
void myFunction() {
  // code to be executed
}
```

Example Explained

- myFunction() is the name of the function
- void means that the function doesn't have return value. You will learn more about return values later in the next chapter
- inside the function (the body), add code that defines what the function should do

FUNCTION DECLARATION AND DEFINITION

A C++ function consist of two parts:

- **Declaration:** the function's name, return type, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

```
void myFunction() { // declaration
  // the body of the function (definition)
}
```

Example

```
// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
```

```
}
```

Example

```
void myFunction() {
  cout<< "I just got executed!\n";
}
```

FUNCTION DEFINITION

C++ function definition consists of a function header and a function body. Here are all the parts of a function

- **Return Type**

  A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

- **Function Name**

  This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- **Parameters**

  A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body**

  The function body contains a collection of statements that define what the function does.

CALL A FUNCTION

- Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

- To call a function, write the function's name followed by two parentheses () and a semicolon ;

- In the following example, myFunction() is used to print a text (the action), when it is called:

Example

Inside main, call myFunction():

```
void myFunction() {
  cout<< "I just got executed!\n";
}

int main() {
  myFunction();
  myFunction();
  myFunction();
  return 0;
}
```

Unit II

Classes and Objects – Constructors and Destructors – New Operator – Operator Overloading and Type Conversions

CLASSES AND OBJECTS

Class: A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.
For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Syntax

```
class Classname
  {
  // Access Modifier
  // data member
  // member function
  };
```

EXAMPLE FOR Class In C++

```
classTest
{
private:
int data1;
float data2;

public:
void function1()
{
```

```
data1 =2;}

float function2()
{
        data2 =3.5;
return data2;
}
};
```

ACCESS MODIFIERS IN C++

- Access modifiers are used to implement an important feature of Object-Oriented Programming known a data hiding
- Access Modifiers or Access Specifiers in a classare used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

1.Public:

All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Example

```
#include<iostream>

using namespace std;


// class definition

class Circle

{
```

```
   public:
      double radius;


double  compute_area();


};
double Circle::compute_area()
{
   radius=4;
std::cout<< radius*radius << std::endl;
}
int main()
{
   Circle circle;
circle.compute_area();
}
```

2.private

The class members declared as *private* can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

Example:

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


// class definition
class Circle
{
   private:  //private data member
```

```
    double radius;
  public:
double  compute_area();


};
double Circle::compute_area()
{
  radius=4;
std::cout<< radius*radius << std::endl;
}
int main()
{
  Circle circle;
circle.compute_area();
}
```

3.Protected

Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

Example

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

// base class
class Parent {

  // protected data members
protected:
  int id_protected;
};

// sub class or derived class
class Child : public Parent {
```

```
public:
   void setId(int id)
   {

      // Child class is able to access the inherited
      // protected data members of the base class

      id_protected = id;
   }

   void displayId()
   {
      cout<< "id_protected is: "
          <<id_protected<<endl;
   }
};

// main function
int main()
{

   Child obj1;

   // member function of the derived class can
   // access the protected data members of the base class

   obj1.setId(81);
   obj1.displayId();
   return 0;
}
float function2()
{
          data2 =3.5;
return data2;
}
};

intmain()
{
Test t;

}
```

CLASSNAME OBJECTVARIABLENAME;

Example

```
classTest
{
private:
int data1;
float data2;

public:
void function1()
{
 data1 =2;
}
```

Example: Object and Class in C++ Programming

Program to illustrate the working of objects and class in C++ Programming

```
#include<iostream>
usingnamespace std;

classTest
{
private:
int data1;
float data2;

public:

voidinsertIntegerData(int d)
{
        data1 = d;
cout<<"Number: "<< data1;
}

floatinsertFloatData()
{
cout<<"\nEnter data: ";
cin>> data2;
return data2;
}
};

intmain()
```

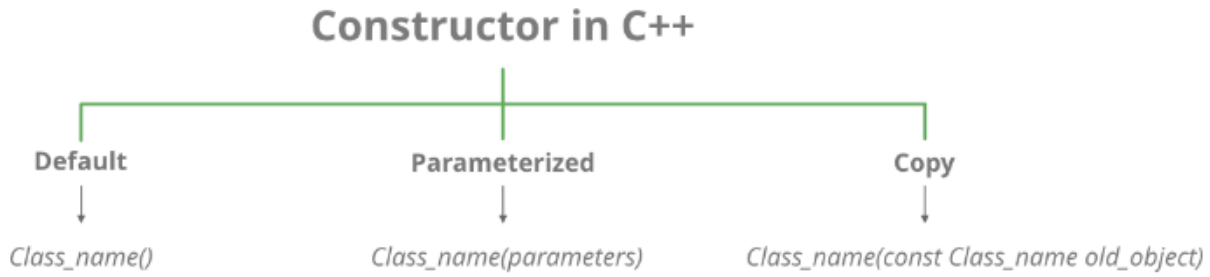
```
{
Test o1, o2;
float secondDataOfObject2;

    o1.insertIntegerData(12);
    secondDataOfObject2 = o2.insertFloatData();

cout<<"You entered "<< secondDataOfObject2;
return0;
}
```

Output
Number: 12
Enter data: 23.3
You entered 23.3

CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS

A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.



constructors are different from a normal function

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us

Types of Constructors

- Default Constructor
- Parameterized Constructor
- Copy Constructor

Default Constructors:

Default constructor is the constructor which doesn't take any argument. It has no parameters.

EXAMPLE FOR DEFAULT CONSTRUCTORS:

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

class construct {
public:
   int a, b;

   // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
   }
};

int main()
{
   // Default constructor called automatically
   // when the object is created
   construct c;
   cout<< "a: " <<c.a<<endl
       << "b: " <<c.b;
   return 1;
}
```

Output:
a: 10
b: 20

Parameterized Constructor

Parameterized Constructor is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

EXAMPLE FOR PARAMETERIZED CONSTRUCTOR

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

class Point {
private:
   int x, y;

public:
   // Parameterized Constructor
   Point(int x1, int y1)
   {
      x = x1;
      y = y1;
   }

   int getX()
   {
      return x;
   }
   int getY()
   {
      return y;
   }
};

int main()
{
   // Constructor called
   Point p1(10, 15);

   // Access values assigned by constructor
   cout<< "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

   return 0;
}
```

Output:

p1.x = 10, p1.y = 15

Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.

3. we have more than one constructors in a class is called Constructor Overloading.

Copy Constructor:

- A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on Copy Constructor.
- Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

Example for copy constructor

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

class Point {
private:
  double x, y;

public:

  Point (double px, double py)  // Non-default Constructor & default Constructor
{
    x = px, y = py;
  }

Point(const Point &p2)
 {
     X=p2.x;
     Y =p2.y;
  }
  double getx( )
   {
      return x;
     }
  double gety( )
  {
    return y;
   }
};

int main(void)
{
```

```
Point p1(10,5)
Point p2=p1;
cout<< ”p1.x” <<p1.getx();
cout<< ”p1.y” <<p1.gety();
cout<<”p2.x”<< p2.getx();
cout<<”p2.y”<<p2.gety();
```

Output:
p1.x=10,p2=15
p1.x=10,p2=15

DESTRUCTORS in C++

A destructor is a special member function that works just opposite to constructor, unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.

Syntax of Destructor

```
~class_name()
{
//Some code
}
```

USES OF DESTRUCTORS in C++

A destructor is **automatically called** when:
1) The program finished execution.
2) When a scope (the { } parenthesis) containing local variable ends.
3) When you call the delete operator.

Destructor Example

```
#include<iostream>
usingnamespace std;
classHelloWorld{
public:
//Constructor
HelloWorld(){
cout<<"Constructor is called"<<endl;
```

```
}
//Destructor
~HelloWorld(){
cout<<"Destructor is called"<<endl;
}
//Member function
voiddisplay(){
cout<<"Hello World!"<<endl;
}
};
intmain(){
//Object created
HelloWorld obj;
//Member function called
obj.display();
return0;
}
```

Output:

Constructoris called

HelloWorld!

Destructoris called

Write The Difference Between Constructor And Member Function

SNO	CONSTRUCTOR	MEMBER FUNCTION
1	Constructor doesn't have a return type	Member function has a return type

2	Constructor is automatically called when we create the object of the class	Member function needs to be called explicitly using object of class
3	When we do not create any constructor in our class, C++ compiler generates a default constructor and insert it into our code.	The same does not apply to member functions.

NEW OPERATOR

Dynamic memory allocation in C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack.**

Applications of new operator

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

Memory allocated vs Normal variables

For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated. For dynamically allocated memory like "int *p = new int[10]", it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes memory leak (memory is not deallocated until program terminates).

Memory allocated/deallocated in C++

C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and**delete** that perform the task of allocating and freeing the memory in a better and easier way. The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax

```
pointer-variable = new data-type;
```

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class. Example:

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;
```

Initialize memory: We can also initialize the memory using new operator:

```
pointer-variable = new data-type(value);
```

Example:

```
int *p = new int(25);
float *q = new float(75.25);
```

Allocate block of memory: new operator is also used to allocate a block(an array) of memory of type *data-type*.

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

- where size(a variable) specifies the number of elements in an array.

Example

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

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.

Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

DELETE OPERATOR

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

```
// Release memory pointed by pointer-variable
delete pointer-variable;
```

Here, pointer-variable is the pointer that points to the data object created by *new*.

Examples:

```
delete p;
```

```
delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use following form of *delete*:

```
// Release block of memory
// pointed by pointer-variable
delete[] pointer-variable;


Example:
   // It will free the entire array
   // pointed by p.
   delete[] p;
```

Example Program

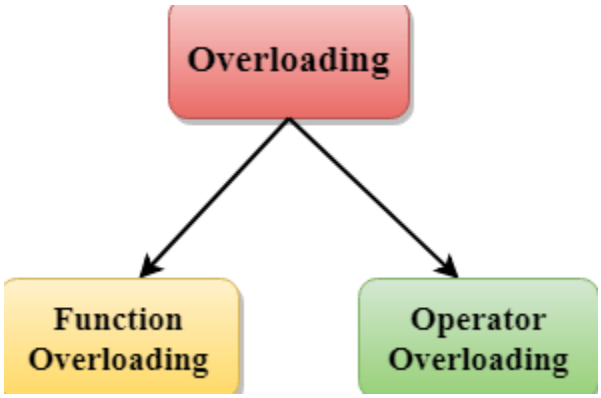
```
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;


int main ()
{
  // Pointer initialization to null
  int* p = NULL;


  // Request memory for the variable
  // using new operator
  p = new(nothrow) int;
  if (!p)
          cout << "allocation of memory failed\n";
  else
```

```
{
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
}

// Request block of memory
// using new operator
float *r = new float(75.25);

cout << "Value of r: " << *r << endl;

// Request block of memory of size n
int n = 5;
int *q = new(nothrow) int[n];

if (!q)
        cout << "allocation of memory failed\n";
else
{
        for (int i = 0; i < n; i++)
                q[i] = i+1;

        cout << "Value store in block of memory: ";
        for (int i = 0; i < n; i++)
```

```
                cout << q[i] << " ";
}

// freed the allocated memory
delete p;
delete r;

    // freed the block of allocated memory
    delete[] q;

    return 0;
}
```

OVERLOADING

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type.



Operators Overloading

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type.
For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.

Types of Operator

- Unary operator loading
- Binary operator loading
- Binary operator overloading using Friend Function

Rules for Operator Overloading

- In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.
- In the case of a friend function, the binary operator should have only two argument and unary should have only one argument.
- All the class member object should be public if operator overloading is implemented.
- Operators that cannot be overloaded are **. .* :: ?:**
- Operator cannot be used to overload when declaring that function as friend function **= () [] ->**.

Syntax

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Example for Unary operator overloading

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

class Complex {
private:
        int real, imag;
public:
        Complex(int r = 0, int i =0)
{
real = r;
imag = i;
}
```

```
        // This is automatically called when '+' is used with
        // between two Complex objects
        Complex operator + (Complex const &obj) {
                Complex res;
                res.real = real + obj.real;
                res.imag = imag + obj.imag;
                return res;
        }
        void print()
{
cout<< real << " + i" <<imag<<endl;
}
};

int main()
{
        Complex c1(10, 5), c2(2, 4);
        Complex c3 = c1 + c2; // An example call to "operator+"
        c3.print();
}
```

Output

```
3.1 + i1.5
1.2 + i2.2
4.3 + i3.7
```

Overloading Binary Operator: In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands.

Example for Binary operator overloading

```
// C++ program to show binary operator overloading
#include <iostream>

using namespace std;

class Distance {
public:
        // Member Object
        int feet, inch;
        // No Parameter Constructor
        Distance()
        {
```

```
            this->feet = 0;
            this->inch = 0;
        }

        // Constructor to initialize the object's value
        // Parametrized Constructor
        Distance(int f, int i)
        {
            this->feet = f;
            this->inch = i;
        }

        // Overloading (+) operator to perform addition of
        // two distance object
        Distance operator+(Distance& d2) // Call by reference
        {
            // Create an object to return
            Distance d3;

            // Perform addition of feet and inches
            d3.feet = this->feet + d2.feet;
            d3.inch = this->inch + d2.inch;

            // Return the resulting object
            return d3;
        }
};

// Driver Code
int main()
{
        // Declaring and Initializing first object
        Distance d1(8, 9);

        // Declaring and Initializing second object
        Distance d2(10, 2);

        // Declaring third object
        Distance d3;

        // Use overloaded operator
        d3 = d1 + d2;
```

```
	// Display the result
	cout<< "\nTotal Feet & Inches: " << d3.feet<< "'" << d3.inch;
	return 0;
}
```

Overloading Binary Operator using a Friend function: In this approach, the operator overloading function must precede with friend keyword, and declare a function class scope. Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator. All the working and implementation would same as binary operator function except this function will be implemented outside of the class scope.
Let's take the same example using the friend function.

// C++ program to show binary operator overloading

```
#include <iostream>

using namespace std;

class Distance {
public:

	// Member Object
	int feet, inch;

	// No Parameter Constructor
	Distance()
	{
		this->feet = 0;
		this->inch = 0;
	}

	// Constructor to initialize the object's value
	// Parametrized Constructor
	Distance(int f, int i)
	{
		this->feet = f;
		this->inch = i;
	}

	// Declaring friend function using friend keyword
	friend Distance operator+(Distance&, Distance&);
};
```

```
// Implementing friend function with two parameters
Distance operator+(Distance& d1, Distance& d2) // Call by reference
{
	// Create an object to return
	Distance d3;

	// Perform addition of feet and inches
	d3.feet = d1.feet + d2.feet;
	d3.inch = d1.inch + d2.inch;

	// Return the resulting object
	return d3;
}

// Driver Code
int main()
{
	// Declaring and Initializing first object
	Distance d1(8, 9);

	// Declaring and Initializing second object
	Distance d2(10, 2);

	// Declaring third object
	Distance d3;

	// Use overloaded operator
	d3 = d1 + d2;

	// Display the result
	cout<< "\nTotal Feet & Inches: " << d3.feet<< "'" << d3.inch;
	return 0;
}
```

Can we overload all operators?

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

```
. (dot)
```

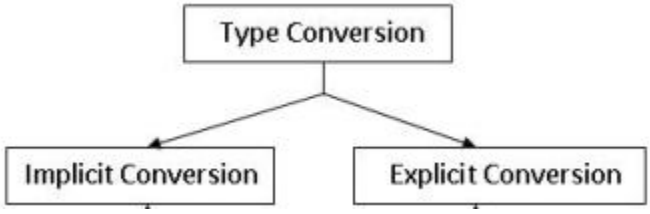
```
   ::
   ?:
sizeof
```

Type Conversion in C++

A type cast is basically a conversion from one type of data to another type.

```
int x;
float x = 3.14;
m=x;
```

There are two types of type conversion:



Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.

Implicit Type Conversion Also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.
- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example Implicit Conversion

```
#include <iostream>
usingnamespacestd;

intmain()
{
   intx = 10; // integer x
   chary = 'a'; // character c
```

```
    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    // x is implicitly converted to float
    floatz = x + 1.0;

    cout<< "x = "<< x <<endl
        << "y = "<< y <<endl
        << "z = "<< z <<endl;

    return0;
}
```

Explicit Type Conversion: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.
In C++, it can be done by two ways:

1. Converting by assignment

2. Conversion using Cast operator

- **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Syntax

```
(type) expression
```

where *type* indicates the data type to which the final result is converted.

Example:

```
#include <iostream>
usingnamespacestd;

intmain()
{
    doublex = 1.2;

    // Explicit conversion from double to int
    intsum = (int)x + 1;

    cout<< "Sum = "<< sum;

    return0;
}
```

Output:

Sum = 2

Conversion using Cast operator: A Cast operator is an **unary operator** which forces one data type to be converted into another data type.

C++ supports four types of casting

- **Static Cast**
- **Dynamic Cast**
- **Const Cast**
- **Reinterpret Cast**

Example

```
#include <iostream>
using namespace std;
int main()
{
   float f = 3.5;

   // using cast operator
   int b = static_cast<int>(f);

   cout<< b;
}
```

Output:

3

Unit II

Classes and Objects – Constructors and Destructors – New Operator – Operator Overloading and Type Conversions

CLASSES AND OBJECTS

Class: A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.
For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Syntax

```
class Classname
  {
  // Access Modifier
  // data member
  // member function
  };
```

EXAMPLE FOR Class In C++

```
class Test
{
   private:
      int data1;
      float data2;

   public:
      void function1()
      {
       data1 = 2;  }

      float function2()
      {
         data2 = 3.5;
         return data2;
      }
  };
```

ACCESS MODIFIERS IN C++

- Access modifiers are used to implement an important feature of Object-Oriented Programming known a data hiding
- Access Modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

4. **Public**
5. **Private**
6. **Protected**

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

1.Public:

All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Example

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

// class definition
class Circle
{
   public:
       double radius;

       double  compute_area();

};
double Circle::compute_area()
{
   radius=4;
   std::cout << radius*radius << std::endl;
}
int main()
{
   Circle circle;
   circle.compute_area();
}
```

2.private

The class members declared as *private* can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

Example:

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

// class definition
class Circle
{
   private:  //private data member
      double radius;
   public:
      double  compute_area();

};
double Circle::compute_area()
{
   radius=4;
   std::cout << radius*radius << std::endl;
}
int main()
{
   Circle circle;
   circle.compute_area();
}
```

3.Protected

Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

Example

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

```
// base class
class Parent {

   // protected data members
protected:
   int id_protected;
};

// sub class or derived class
class Child : public Parent {

public:
   void setId(int id)
   {

      // Child class is able to access the inherited
      // protected data members of the base class

      id_protected = id;
   }

   void displayId()
   {
      cout << "id_protected is: "
           << id_protected << endl;
   }
};

// main function
int main()
{

   Child obj1;

   // member function of the derived class can
   // access the protected data members of the base class

   obj1.setId(81);
   obj1.displayId();
   return 0;
}
float function2()
```

```
      {
         data2 = 3.5;
         return data2;
      }
  };

int main()
{
   Test t;

}
```

CLASSNAME OBJECTVARIABLENAME;

Example

```
class Test
{
   private:
      int data1;
      float data2;

   public:
      void function1()
      {
       data1 = 2;
}
```

Example: Object and Class in C++ Programming

Program to illustrate the working of objects and class in C++ Programming

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

class Test
{
   private:
      int data1;
      float data2;

   public:

      void insertIntegerData(int d)
      {
```

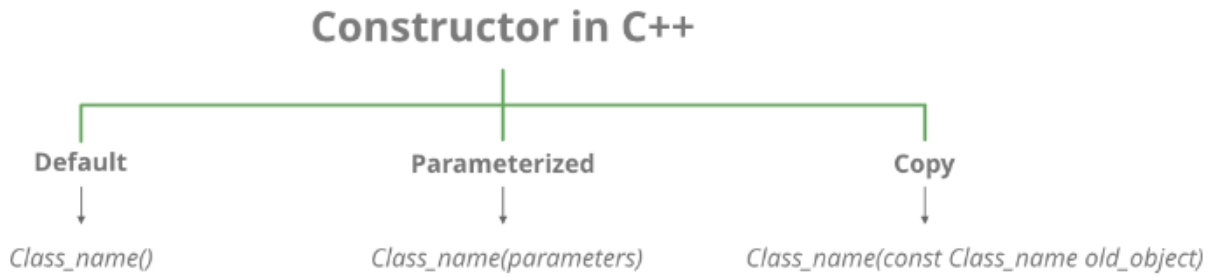
```
        data1 = d;
        cout << "Number: " << data1;
      }

     float insertFloatData()
     {
        cout << "\nEnter data: ";
        cin >> data2;
        return data2;
      }
};

int main()
{
    Test o1, o2;
    float secondDataOfObject2;

    o1.insertIntegerData(12);
    secondDataOfObject2 = o2.insertFloatData();

    cout << "You entered " << secondDataOfObject2;
    return 0;
}
```

Output
Number: 12
Enter data: 23.3
You entered 23.3

CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS

A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.



constructors are different from a normal function

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us

Types of Constructors

- **Default Constructor**
- **Parameterized Constructor**
- **Copy Constructor**

Default Constructors:

Default constructor is the constructor which doesn't take any argument. It has no parameters.

EXAMPLE FOR DEFAULT CONSTRUCTORS:

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

class construct {
public:
   int a, b;

   // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
   }
};

int main()
```

```
{
   // Default constructor called automatically
   // when the object is created
   construct c;
   cout << "a: " << c.a << endl
        << "b: " << c.b;
   return 1;
}
```

Output:
a: 10
b: 20

Parameterized Constructor

Parameterized Constructor is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

EXAMPLE FOR PARAMETERIZED CONSTRUCTOR

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

class Point {
private:
   int x, y;

public:
   // Parameterized Constructor
   Point(int x1, int y1)
   {
      x = x1;
      y = y1;
   }

   int getX()
   {
      return x;
   }
   int getY()
   {
```

```
        return y;
    }
};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

Output:

p1.x = 10, p1.y = 15

Uses of Parameterized constructor:

4. It is used to initialize the various data elements of different objects with different values when they are created.
5. It is used to overload constructors.
6. we have more than one constructors in a class is called Constructor Overloading.

Copy Constructor:

- A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on Copy Constructor.
- Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

Example for copy constructor

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

class Point {
private:
  double x, y;

public:
```

```
 Point (double px, double py)  // Non-default Constructor & default Constructor
{
   x = px, y = py;
 }

Point(const Point &p2)
{
     X=p2.x;
     Y =p2.y;
 }
 double getx( )
  {
      return x;
     }
 double gety( )
 {
    return y;
   }
};

int main(void)
{
 Point p1(10,5)
 Point p2=p1;
cout<< ”p1.x” <<p1.getx();
cout<< ”p1.y” <<p1.gety();
 cout<<”p2.x”<< p2.getx();
 cout<<”p2.y”<<p2.gety();
```

Output:
p1.x=10,p2=15
p1.x=10,p2=15

DESTRUCTORS in C++

A destructor is a special member function that works just opposite to constructor, unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.

Syntax of Destructor

```
~class_name()

{
```

```
    //Some code
}
```

USES OF DESTRUCTORS in C++

A destructor is **automatically called** when:
1) The program finished execution.
2) When a scope (the { } parenthesis) containing local variable ends.
3) When you call the delete operator.

Destructor Example

```
#include <iostream>
using namespace std;
class HelloWorld{
public:
  //Constructor
  HelloWorld(){
    cout<<"Constructor is called"<<endl;
  }
  //Destructor
  ~HelloWorld(){
    cout<<"Destructor is called"<<endl;
   }
   //Member function
   void display(){
     cout<<"Hello World!"<<endl;
   }
};
int main(){
   //Object created
   HelloWorld obj;
   //Member function called
   obj.display();
```

```
  return 0;
}
```

Output:

Constructor is called

Hello World!

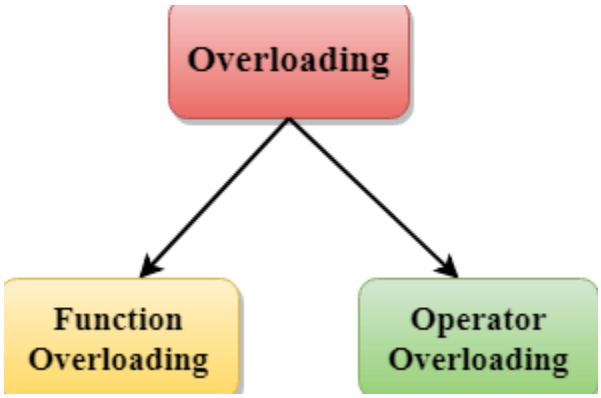
Destructor is called

Write The Difference Between Constructor And Member Function

SNO	CONSTRUCTOR	MEMBER FUNCTION
1	Constructor doesn't have a return type	Member function has a return type
2	Constructor is automatically called when we create the object of the class	Member function needs to be called explicitly using object of class
3	When we do not create any constructor in our class, C++ compiler generates a default constructor and insert it into our code.	The same does not apply to member functions.

Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type.



Operators Overloading

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type.
For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.

Types of Operator

- Unary operator loading
- Binary operator loading
- Binary operator overloading using Friend Function

Rules for Operator Overloading

- In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.
- In the case of a friend function, the binary operator should have only two argument and unary should have only one argument.
- All the class member object should be public if operator overloading is implemented.
- Operators that cannot be overloaded are **. .* :: ?:**
- Operator cannot be used to overload when declaring that function as friend function **= () [] ->**.

Syntax

```
return_type class_name  : : operator op(argument_list)
{
   // body of the function.
}
```

Example for Unary operator overloading

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

class Complex {
private:
        int real, imag;
public:
        Complex(int r = 0, int i =0)
        {
```

```
        real = r;
        imag = i;
    }

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
            Complex res;
            res.real = real + obj.real;
            res.imag = imag + obj.imag;
            return res;
    }
    void print()
     {
       cout << real << " + i" << imag << endl;
      }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Output

```
3.1 + i1.5
1.2 + i2.2
4.3 + i3.7
```

Overloading Binary Operator: In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands.

Example for Binary operator overloading

```
// C++ program to show binary operator overloading
#include <iostream>

using namespace std;

class Distance {
public:
    // Member Object
```

```
        int feet, inch;
        // No Parameter Constructor
        Distance()
        {
                this->feet = 0;
                this->inch = 0;
        }

        // Constructor to initialize the object's value
        // Parametrized Constructor
        Distance(int f, int i)
        {
                this->feet = f;
                this->inch = i;
        }

        // Overloading (+) operator to perform addition of
        // two distance object
        Distance operator+(Distance& d2) // Call by reference
        {
                // Create an object to return
                Distance d3;

                // Perform addition of feet and inches
                d3.feet = this->feet + d2.feet;
                d3.inch = this->inch + d2.inch;

                // Return the resulting object
                return d3;
        }
};

// Driver Code
int main()
{
        // Declaring and Initializing first object
        Distance d1(8, 9);

        // Declaring and Initializing second object
        Distance d2(10, 2);

        // Declaring third object
        Distance d3;
```

```
	// Use overloaded operator
	d3 = d1 + d2;

	// Display the result
	cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
	return 0;
}
```

Overloading Binary Operator using a Friend function: In this approach, the operator overloading function must precede with friend keyword, and declare a function class scope. Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator. All the working and implementation would same as binary operator function except this function will be implemented outside of the class scope.
Let's take the same example using the friend function.

// C++ program to show binary operator overloading

```
#include <iostream>

using namespace std;

class Distance {
public:

	// Member Object
	int feet, inch;

	// No Parameter Constructor
	Distance()
	{
		this->feet = 0;
		this->inch = 0;
	}

	// Constructor to initialize the object's value
	// Parametrized Constructor
	Distance(int f, int i)
	{
		this->feet = f;
		this->inch = i;
	}
```

```
        // Declaring friend function using friend keyword
        friend Distance operator+(Distance&, Distance&);
};

// Implementing friend function with two parameters
Distance operator+(Distance& d1, Distance& d2) // Call by reference
{
        // Create an object to return
        Distance d3;

        // Perform addition of feet and inches
        d3.feet = d1.feet + d2.feet;
        d3.inch = d1.inch + d2.inch;

        // Return the resulting object
        return d3;
}

// Driver Code
int main()
{
        // Declaring and Initializing first object
        Distance d1(8, 9);

        // Declaring and Initializing second object
        Distance d2(10, 2);

        // Declaring third object
        Distance d3;

        // Use overloaded operator
        d3 = d1 + d2;

        // Display the result
        cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
        return 0;
}
```

Can we overload all operators?

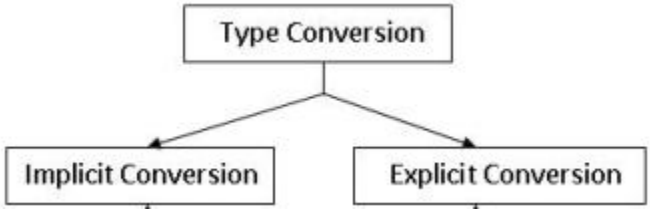
Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

```
   . (dot)
```

```
::
?:
sizeof
```

Type Conversion in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:



Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.

Implicit Type Conversion Also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.
- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example Implicit Conversion

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

int main()
{
   int x = 10; // integer x
   char y = 'a'; // character c

   // y implicitly converted to int. ASCII
   // value of 'a' is 97
   x = x + y;
```

```
    // x is implicitly converted to float
    float z = x + 1.0;

    cout << "x = " << x << endl
         << "y = " << y << endl
         << "z = " << z << endl;

    return 0;
}
```

Explicit Type Conversion: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.
In C++, it can be done by two ways:

1. Converting by assignment

2. Conversion using Cast operator

- **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.
  Syntax:
  ```
  (type) expression
  ```
  where *type* indicates the data type to which the final result is converted.

Example:

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

int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    cout << "Sum = " << sum;

    return 0;
}
```

Output:
Sum = 2

Conversion using Cast operator: A Cast operator is an **unary operator** which forces one data type to be converted into another data type.
C++ supports four types of casting

- **Static Cast**
- **Dynamic Cast**
- **Const Cast**
- **Reinterpret Cast**

Example

```
#include <iostream>
using namespace std;
int main()
{
   float f = 3.5;

   // using cast operator
   int b = static_cast<int>(f);

   cout << b;
}
```

Output:

3

Unit- III

Inheritance: Extending Classes – Pointers – Virtual Functions and Polymorphism

INHERITANCE: EXTENDING CLASSES IN C++

- In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.
- In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.
- In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class.
- The derived class is the specialized class for the base class.

Derived Classes

- A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name : visibility-mode base_class_name
{
   // body of the derived class.
}
```

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class.
- Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.

- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class.
- Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

ADVANTAGE OF C++ INHERITANCE

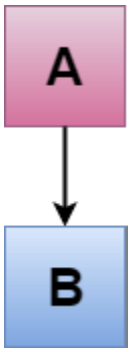
Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So, less code is required in the class.

Types of Inheritance

- **Single inheritance**
- **Multiple inheritance**
- **Hierarchical inheritance**
- **Multilevel inheritance**
- **Hybrid inheritance**

SINGLE INHERITANCE

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

Example Program

```
#include <iostream>

using namespace std;

class Account {

public:

float salary = 60000;

 };

class Programmer: public Account {

public:

float bonus = 5000;

 };

int main(void)

{

Programmer p1;

cout<<"Salary: "<<p1.salary<<endl;

cout<<"Bonus: "<<p1.bonus<<endl;

return 0;

}
```

Output:

```
Salary: 60000
```

Bonus: 5000

Example using Animal Characteristics

```
#include <iostream>
using namespace std;
 class Animal
{
 public:
 void eat()
{
    cout<<"Eating..."<<endl;
 }
 };

class Dog: public Animal
{
 public:
 void bark()
{
    cout<<"Barking...";
  }
};
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output:

```
Eating...
Barking...
```

How to make a Inherit Access Modifier

- The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.
- C++ introduces a third visibility modifier, i.e., **protected**.
- The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

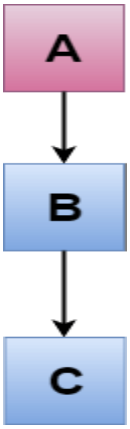
Visibility modes can be classified into three categories:

- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.
- Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

MULTILEVEL INHERITANCE

- Multilevel inheritance **is a process of deriving a class from another derived class.**



- When one class inherits another class which is further inherited by another class, it is known as multi-level inheritance in C++.
- Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Example Program

```
#include <iostream>
using namespace std;
class Animal
{
public:
void eat()
{
cout<<"Eating..."<<endl;
 }
};
class Dog: public Animal
{
public:
void bark()
{
cout<<"Barking..."<<endl;
}
};
classBabyDog: public Dog
{
public:
void weep()
{
```

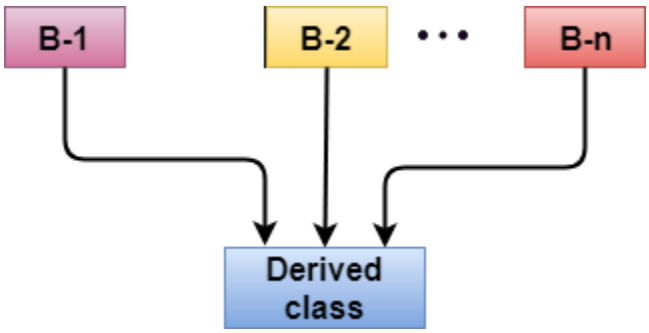
```
cout<<"Weeping...";
 }
};
int main(void) {
BabyDog d1;
d1.eat();
d1.bark();
d1.weep();
return 0;
}
```

Output:

Eating...
Barking...
Weeping...

MULTIPLE INHERITANCE

- **Multiple inheritance** is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

```
class D : visibility B-1, visibility B-2
{
   // Body of the class;
}
```

Example:

```
#include <iostream>
using namespace std;
class A
{
protected:
int a;
public:
voidget_a(int n)
   {
      a = n;
   }
};

class B
{
protected:
int b;
public:
voidget_b(int n)
   {
      b = n;
   }
};
class C : public A,public B
{
```

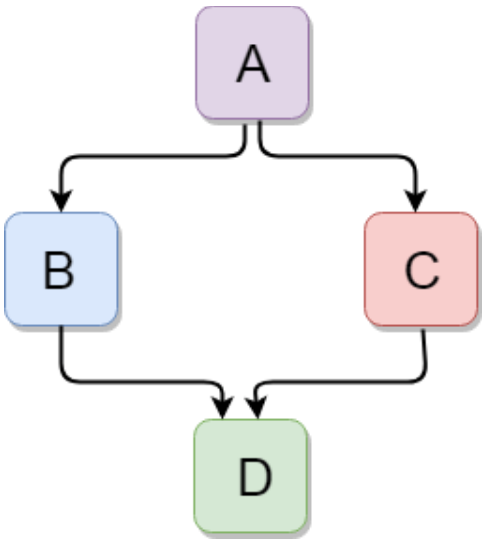
```
public:
void display()
   {
std::cout<< "The value of a is : " <<a<<std::endl;
std::cout<< "The value of b is : " <<b<<std::endl;
cout<<"Addition of a and b is : "<<a+b;
   }
};
int main()
{
  C c;
c.get_a(10);
c.get_b(20);
c.display();
return 0;
}
```

Output:

The value of a is : 10
The value of b is : 20
Addition of a and b is : 30

HYBRID INHERITANCE

- Hybrid inheritance is a combination of more than one type of inheritance.



Example

```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
      std::cout << "Enter the value of 'a' : " << std::endl;
     cin>>a;
   }
};

class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
      std::cout << "Enter the value of 'b' : " << std::endl;
     cin>>b;
```

```
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
         get_a();
         get_b();
         get_c();
         std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    }
};
int main()
{
    D d;
    d.mul();
    return 0;
}
```

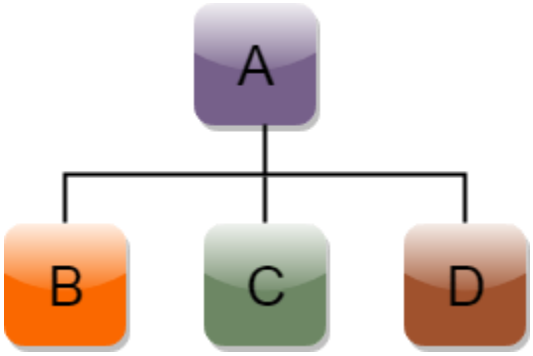
Output:

```
Enter the value of 'a' :
10
```

```
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

HIERARCHICAL INHERITANCE

- Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax of Hierarchical inheritance:

```
class A
{
   // body of the class A.
}
class B : public A
{
  // body of class B.
}
 class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Example Program:

```
#include <iostream>
using namespace std;
class Shape                // Declaration of base class.
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
   }
};
class Triangle : public Shape    // inheriting Shape class
{
   public:
   int triangle_area()
   {
      float result = 0.5*a*b;
      return result;
   }
};
int main()
{
   Rectangle r;
   Triangle t;
   int length, breadth, base, height;
   std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
```

```
   cin>>length>>breadth;
   r.get_data(length,breadth);
   int m = r.rect_area();
   std::cout << "Area of the rectangle is : " <<m<< std::endl;
   std::cout << "Enter the base and height of the triangle: " << std::endl;
   cin>>base>>height;
   t.get_data(base,height);
   float n = t.triangle_area();
   std::cout <<"Area of the triangle is : "  << n<<std::endl;
   return 0;
}
```

Output:

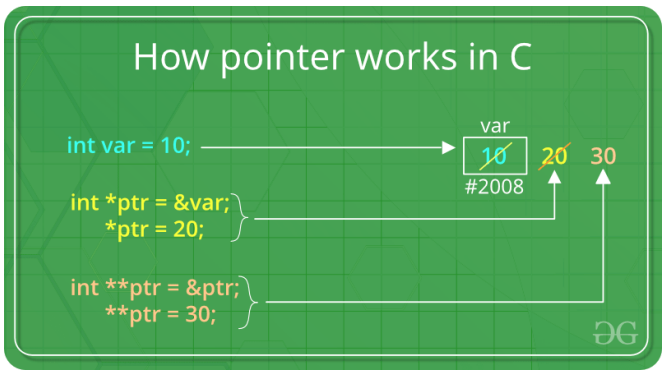
```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

POINTERS

- Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures.

Syntax:

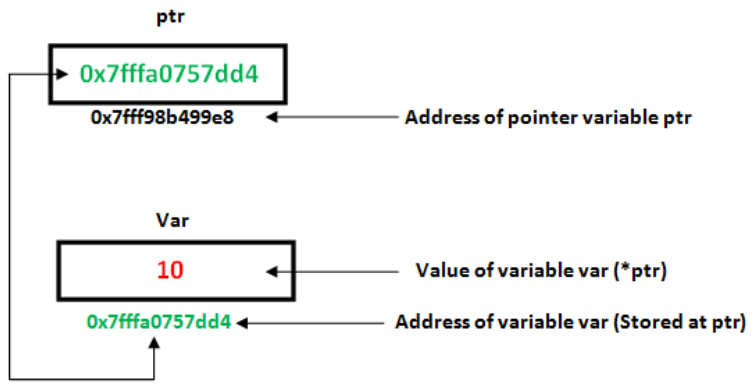
```
datatype *var_name;

int *ptr;   //ptr can point to an address which holds int data
```



How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

The reason we associate data type to a pointer is **that it knows how many bytes the data is stored in**. When we increment a pointer, we increase the pointer by the size of data type to which it points.



Example:

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

```
void geeks()
{
        intvar = 20;

                //declare pointer variable
        int *ptr;

        //note that data type of ptr and var must be same
        ptr = &var;

        // assign the address of a variable to a pointer
        cout<< "Value at ptr = " <<ptr<< "\n";
        cout<< "Value at var = " <<var<< "\n";
        cout<< "Value at *ptr = " << *ptr<< "\n";
}
//Driver program
int main()
{
        geeks();
}
```

Output:

```
Value at ptr = 0x7ffcb9e9ea4c

Value at var = 20

Value at *ptr = 20
```

REFERENCES AND POINTERS

There are 3 ways to pass C++ arguments to a function:

- call-by-value
- call-by-reference with pointer argument
- call-by-reference with reference argument

// C++ program to illustrate call-by-methods in C++

```
#include <iostream.h>
```

```
using namespace std;
//Pass-by-Value
int square1(int n)
{
	//Address of n in square1() is not the same as n1 in main()
	cout<< "address of n1 in square1(): " <<&n << "\n";

	// clone modified inside the function
	n *= n;
	return n;
}
//Pass-by-Reference with Pointer Arguments
void square2(int *n)
{
	//Address of n in square2() is the same as n2 in main()
	cout<< "address of n2 in square2(): " << n << "\n";

	// Explicit de-referencing to get the value pointed-to
	*n *= *n;
}
//Pass-by-Reference with Reference Arguments
void square3(int&n)
{
	//Address of n in square3() is the same as n3 in main()
	cout<< "address of n3 in square3(): " <<&n << "\n";

	// Implicit de-referencing (without '*')
```

```
        n *= n;
}
void geeks()
{
        //Call-by-Value
        int n1=8;
        cout<< "address of n1 in main(): " <<&n1 << "\n";
        cout<< "Square of n1: " << square1(n1) << "\n";
        cout<< "No change in n1: " << n1 << "\n";

        //Call-by-Reference with Pointer Arguments
        int n2=8;
        cout<< "address of n2 in main(): " <<&n2 << "\n";
        square2(&n2);
        cout<< "Square of n2: " << n2 << "\n";
        cout<< "Change reflected in n2: " << n2 << "\n";

        //Call-by-Reference with Reference Arguments
        int n3=8;
        cout<< "address of n3 in main(): " <<&n3 << "\n";
        square3(n3);
        cout<< "Square of n3: " << n3 << "\n";
        cout<< "Change reflected in n3: " << n3 << "\n";
}
//Driver program
int main()
{
```

```
    geeks();
}
```

Output:

```
address of n1 in main(): 0x7ffcdb2b4a44
address of n1 in square1(): 0x7ffcdb2b4a2c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7ffcdb2b4a48
address of n2 in square2(): 0x7ffcdb2b4a48
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7ffcdb2b4a4c
address of n3 in square3(): 0x7ffcdb2b4a4c
Square of n3: 64
Change reflected in n3: 64
```

- In C++, by default arguments are passed by value and the changes made in the called function will not reflect in the passed variable.
- The changes are made into a clone made by the called function. If wish to modify the original copy directly (especially in passing huge object or array) and/or avoid the overhead of cloning, we use pass-by-reference.
- Pass-by-reference with Reference Arguments does not require any clumsy syntax for referencing and dereferencing.

- Function pointers in C
- Pointer to a function

ARRAY NAME AS POINTERS

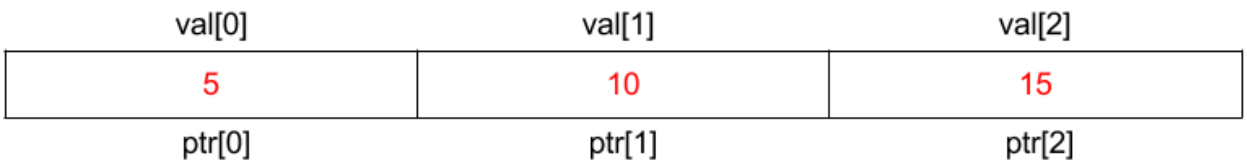
- An array name contains the address of first element of the array which acts like constant pointer. It means, the address stored in array name can't be changed. **For example**, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

Example

```
#include <iostream>
using namespace std;
void geeks()
{
        //Declare an array
        int val [3] = { 5, 10, 20 };

        //declare pointer variable
        int *ptr;

        //Assign the address of val[0] to ptr
        // We can use ptr=&val[0];(both are same)
        ptr = val ;
        cout<< "Elements of the array are: ";
        cout<<ptr [0] << " " <<ptr [1] << " " <<ptr [2];
}
//Driver program
int main()
{
        geeks();
}
```

Output:

Elements of the array are: 5 10 20



- If pointer ptr is sent to a function as an argument, the array val can be accessed in a similar fashion.

Pointer Expressions and Pointer Arithmetic

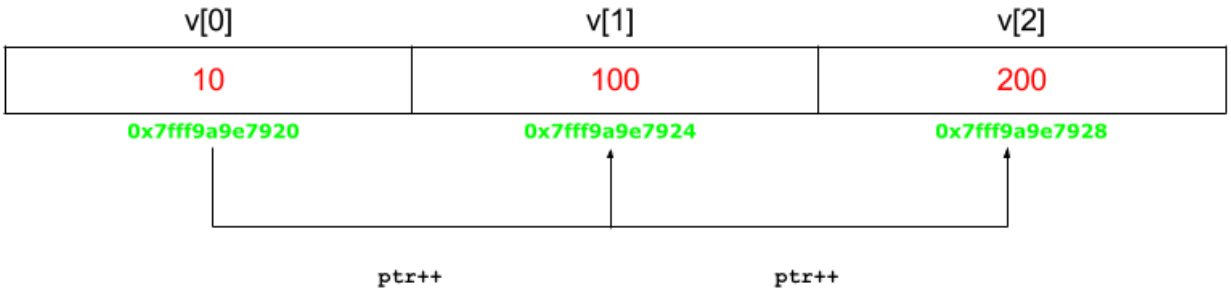
A limited set of arithmetic operations can be performed on pointers which are:

- incremented (++)
- decremented (—)
- an integer may be added to a pointer (+ or +=)
- an integer may be subtracted from a pointer (– or -=)
- difference between two pointers (p1-p2)

```
// C++ program to illustrate Pointer Arithmetic in C++
#include <bits/stdc++.h>
using namespace std;
void geeks()
{
        //Declare an array
        int v[3] = {10, 100, 200};

        //declare pointer variable
        int *ptr;

        //Assign the address of v[0] to ptr
        ptr = v;

        for (inti = 0; i< 3; i++)
        {
                cout<< "Value at ptr = " <<ptr<< "\n";
                cout<< "Value at *ptr = " << *ptr<< "\n";

                // Increment pointer ptr by 1
```

```
                ptr++;
        }
}

//Driver program
int main()
{
        geeks();
}
```

```
Output:
Value at ptr = 0x7fff9a9e7920
Value at *ptr = 10
Value at ptr = 0x7fff9a9e7924
Value at *ptr = 100
Value at ptr = 0x7fff9a9e7928
Value at *ptr = 200
```



Advanced Pointer Notation

- Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration

```
int nums[2][3]  =  { { 16, 18, 20 }, { 25, 26, 27 } };
```

In general, nums[i][j] is equivalent to *(*(nums+i)+j)

Pointer Notation	Array Notation	Value
*(*nums)	nums[0] [0]	16
*(*nums+1)	nums[0] [1]	18
*(*nums+2)	nums[0] [2]	20
((nums + 1))	nums[1] [0]	25
((nums + 1)+1)	nums[1] [1]	26
((nums + 1)+2)	nums[1] [2]	27

Pointers and String literals

String literals are arrays containing null-terminated character sequences. String literals are arrays of type character plus terminating null-character, with each of the elements being of type const char (as characters of string can't be modified).

```
const char * ptr = "geek";
```

This declares an array with the literal representation for "geek", and then a pointer to its first element is assigned to ptr. If we imagine that "geek" is stored at the memory locations that start at address 1800, we can represent the previous declaration as:

'g'	'e'	'e'	'k'	'\0'
1800	1801	1802	1803	1804

As pointers and arrays behave in the same way in expressions, ptr can be used to access the characters of string literal. For example:

```
char x = *(ptr+3);
char y = ptr[3];
```

Here, both x and y contain k stored at 1803 (1800+3).

POINTERS TO POINTERS

- In C++, we can create a pointer to a pointer that in turn may point to data or other pointer.
- The syntax simply requires the unary operator (*) for each level of indirection while declaring the pointer.

```
char a;
char *b;
char ** c;
a = ’g’;
b = &a;
c = &b;
```

- Here b points to a char that stores ‘g’ and c points to the pointer b.

VOID POINTERS

- This is a special type of pointer available in C++ which represents absence of type.
- void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).
- This means that void pointers have great flexibility as it can point to any data type. There is payoff for this flexibility.
- These pointers cannot be directly dereferenced.
- They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.

```
#include <iostream.h>

using namespace std;

void increase(void *data,int ptrsize)

{

        if(ptrsize == sizeof(char))

        {

                char *ptrchar;
```

```
            //Typecast data to a char pointer
            ptrchar = (char*)data;

            //Increase the char stored at *ptrchar by 1
            (*ptrchar)++;
            cout<< "*data points to a char"<<"\n";
        }
        else if(ptrsize == sizeof(int))
        {
            int *ptrint;

            //Typecast data to aint pointer
            ptrint = (int*)data;

            //Increase the int stored at *ptrchar by 1
            (*ptrint)++;
            cout<< "*data points to an int"<<"\n";
        }
}
void geek()
{
```

```
        // Declare a character
        char c='x';


        // Declare an integer
        inti=10;


        //Call increase function using a char and int address respectively
        increase(&c,sizeof(c));
        cout<< "The new value of c is: " << c <<"\n";
        increase(&i,sizeof(i));
        cout<< "The new value of i is: " <<i<<"\n";


}
//Driver program
int main()
{
        geek();
}
```

Output:

*data points to a char

The new value of c is: y

*data points to an int

```
The new value of i is: 11
```

Invalid pointers

- A pointer should point to a valid address but not necessarily to valid elements (like for arrays). These are called invalid pointers. Uninitialized pointers are also invalid pointers.

```
int *ptr1;
int arr[10];
int *ptr2 = arr+20;
```

- Here, ptr1 is uninitialized so it becomes an invalid pointer and ptr2 is out of bounds of arr so it also becomes an invalid pointer. (Note: invalid pointers do not necessarily raise compile errors)

NULL Pointers

- Null pointer is a pointer which point nowhere and not just an invalid address.
- Following are 2 methods to assign a pointer as NULL;

```
int *ptr1 = 0;
int *ptr2 = NULL;
```

this POINTER

- Every object in C++ has access to its own address through an important pointer called **this** pointer.
- The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.
- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Example

```
#include <iostream>

using namespace std;

class Box {
   public:
      // Constructor definition
```

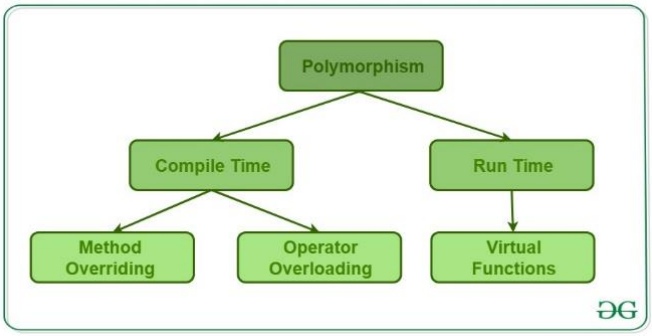
```
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume() {
         return length * breadth * height;
      }
      int compare(Box box) {
         return this->Volume() > box.Volume();
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

int main(void) {
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   if(Box1.compare(Box2)) {
      cout << "Box2 is smaller than Box1" <<endl;
   } else {
      cout << "Box2 is equal to or larger than Box1" <<endl;
   }

   return 0;
}
```

POLYMORPHISM IN C++

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.
- Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



Compile time polymorphism:

- The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding.
- Now, let's consider the case where function name and prototype is same

Types of compile time polymorphism:

- Function Overloading
- Operator overloading

Function Overloading:

- When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**.
- Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

Rules of Function Overloading

1) Function declarations that differ only in the return type.
2) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.

3) Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types.
4) Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.
5) Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called.

Example:

```
#include <iostream.h>

using namespace std;
class Geeks
{
        public:

        // function with 1 int parameter
        void func(int x)
        {
                cout<< "value of x is " << x <<endl;
        }

        // function with same name but 1 double parameter
        void func(double x)
        {
                cout<< "value of x is " << x <<endl;
        }

        // function with same name and 2 int parameters
        void func(int x, int y)
        {
                cout<< "value of x and y is " << x << ", " << y <<endl;
        }
```

```
};

int main() {

        Geeks obj1;

        // Which function is called will depend on the parameters passed
        // The first 'func' is called
        obj1.func(7);

        // The second 'func' is called
        obj1.func(9.132);

        // The third 'func' is called
        obj1.func(85,64);
        return 0;
}
```

Output

```
value of x is 7

value of x is 9.132

value of x and y is 85, 64
```

Operator Overloading:

- C++ also provide option to overload operators.
- For example, we can make the operator (‘+’) for string class to concatenate two strings.
- We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands, adds them and when placed between string operands, concatenates them.

Example

```
// Operator Overloading

#include<iostream>

using namespace std;
```

```
class Complex {
private:
	int real, imag;
public:
	Complex(int r = 0, int i =0) {real = r; imag = i;}

	// This is automatically called when '+' is used with
	// between two Complex objects
	Complex operator + (Complex const &obj) {
		Complex res;
		res.real = real + obj.real;
		res.imag = imag + obj.imag;
		return res;
	}
	void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
	Complex c1(10, 5), c2(2, 4);
	Complex c3 = c1 + c2; // An example call to "operator+"
	c3.print();
}
```

Output:

```
12 + i9
```

- In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers.

VIRTUAL FUNCTION IN C++

- A virtual function is a member function which is declared within a base class and is re-defined (Overriden) by a derived class.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

RULES FOR VIRTUAL FUNCTIONS

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

Example

```
#include <iostream>

using namespace std;
```

```
class base {
public:
	virtual void print()
	{
		cout << "print base class" << endl;
	}

	void show()
	{
		cout << "show base class" << endl;
	}
};

class derived : public base {
public:
	void print()
	{
		cout << "print derived class" << endl;
	}

	void show()
	{
```

```
            cout << "show derived class" << endl;
        }
};

int main()
{
        base* bptr;
        derived d;
        bptr = &d;

        // virtual function, binded at runtime
        bptr->print();

        // Non-virtual function, binded at compile time
        bptr->show();
}
```

Output:

```
print derived class
show base class
```

PURE VIRTUAL FUNCTIONS AND ABSTRACT CLASSES IN C++

- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation.
- Such a class is called abstract class.
- For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw().
- Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.
- A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it.
- A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
// An abstract class
class Test
{
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show() = 0;

  /* Other members */
};
```

Example

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

class Base
{
  int x;
public:
   virtual void fun() = 0;
   int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
```

```
{
   int y;
public:
   void fun() { cout << "fun() called"; }
};

int main(void)
{
   Derived d;
   d.fun();
   return 0;
}
```

Output

```
fun() called
```

DIFFERENCES B/W COMPILE TIME AND RUN TIME POLYMORPHISM

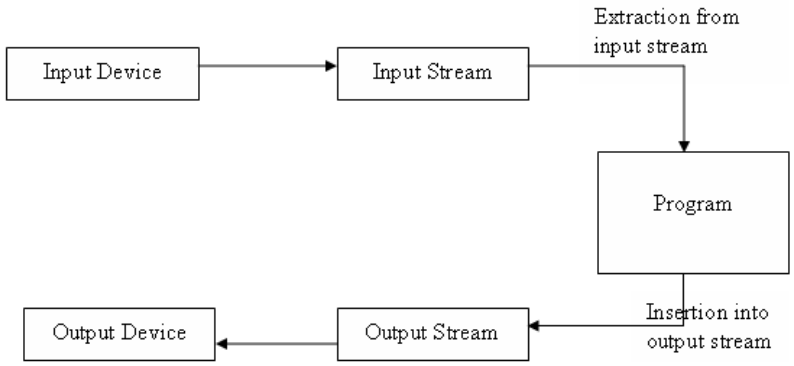
Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.

It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

Unit –IV- Managing Console I/O Operations – Working with Files – Templates – Exception Handling

Basic Input / Output in C++

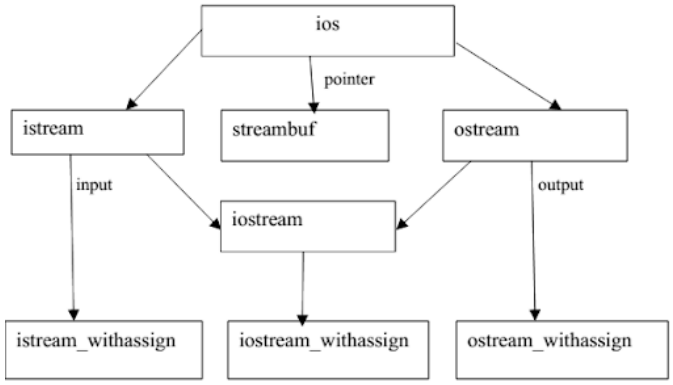
- C++ comes with libraries which provides us with many ways for performing input and output. In C++ input and output is performed in the form of a sequence of bytes or more commonly known as **streams**.
- **Input Stream:** If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.



Header files available in C++ for Input/Output operations are:

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions to objects like cin, cout, cerr etc.
2. **iomanip**: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision etc.

Standard output stream (cout): Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(<<).



Stream classes for console I/O operations

```
#include <iostream>

using namespace std;

int main()
{
    char sample[] = "GeeksforGeeks";

    cout << sample << " - A computer science portal for geeks";

    return 0;
}
```

Output :

```
GeeksforGeeks - A computer science portal for geeks
```

- In the above program the insertion operator(<<) inserts the value of the string variable sample followed by the string “A computer science portal for geeks” in the standard output stream cout which is then displayed on screen.

standard input stream (cin): Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class istream and is used to read input from the standard input device which is usually a keyboard.

- The extraction operator(>>) is used along with the object cin for reading inputs. The extraction operator extracts the data from the object cin which is entered using the keyboard.

Example program

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

int main()
{
	int age;

	cout << "Enter your age:";
	cin >> age;
	cout << "\nYour age is: " << age;

	return 0;
}
```

Input :

```
18
```

Output:

```
Enter your age:

Your age is: 18
```

- The above program asks the user to input the age. The object cin is connected to the input device.
- The age entered by the user is extracted from cin using the extraction operator(>>) and the extracted data is then stored in the variable **age** present on the right side of the extraction operator.

Un-buffered standard error stream (cerr): The C++ cerr is the standard error stream which is used to output the errors. This is also an instance of the ostream class. As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display later.

Example Progam

```
#include <iostream>
```

```
using namespace std;
int main()
{
   cerr << "An error occured";
    return 0;
}
```

Output:

```
An error occurred
```

Buffered standard error stream (clog): This is also an instance of iostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. The error message will be displayed on the screen too.

Example Program

```
#include <iostream>
using namespace std;
int main()
{
   clog << "An error occured";
    return 0;
}
```

Output:

```
An error occurred
```

Types of console I/O operations form:

- Unformatted console input/output operation
- Formatted console input/output operation

Unformatted console input output operations

- These input / output operations are in unformatted mode. The following are operations of unformatted console input / output operations:

get() function

- **The classes istream and ostream defines two member function get() and put() representation to handle the single character input/output operation.**

- **We can use get(char *) and get(void) to fetch a character including the blank space, tab and the newline character.**
- **get(char *) – assign input character to its argument**
- **get(void) – returns the type of input character**

put() function

- **put() function is a member of ostream class, can be used to output a line of text character by character.**

Syntax

```
cout.put(ch);
```

ch – must be character value.

cout.put(68)-can use the number as an argument but display the character (68- D)

Example program

```
#include<iostream>

using namespace std;

int main()

{

 int cout=0;

 char c;

cout<<"INPUT TEXT\n";

cin.get(c);

while(c!='\n')

{

        cout.put(c);
```

```
        cout++;
        cin.get(c);
}
cout<<"\n Number of characters = " <<c<<"\n";
return 0;
}
```

Output:

INPUT TEXT

Object oriented programming

Number of characters = 27

getline()

- getline() reads whole line of text that ends with newline character. This function can be invoked by using the object cin as follows

Syntax:

```
cin.getline(line,size);
```

char name[20];

cin.getline(name,20)

Example:

```
int main()
{
        int size=20;
```

```
        char city[20];

        cout<<"Enter your city name :";

        cin.getline(city,20); //It takes 20 characters as input;

        cout<<"your city name:"<<city<<endl;

        return 0;

}
```

Output

write() function

- write() function displays an entire line

Syntax

```
cout.write(line, size)
```

- The first argument - line – name of the string to be displayed.
- The second argument – size – indicates the number of characters to display.

Example Program

```
#include< iostream>
#include<string>
using namespace std;
int main()
{
        char *string1="c++";
        char *string2="programming";
        int m=strlen(string1);
        int n=strlen(string2);
```

```
        for(int i=1; i<n;i++)
        {
                cout.write(string2,i);
                cout<<"\n";
        }
        for(i=n;i>0;i++)
        {
                cout.write(string2,i);
                cout<<"\n";
        }
        cout.write(string1,m).write(string2,n);
        cout <<"\n";

        cout.write(string1,10);
        return 0;
}
```

Output

P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr

Formatted console I/O operation

C++ supports a number of features that could be used for formatting the output. These features include:

- **ios class functions and flags**
- **Manipulators**
- **User-defined output functions**

ios class function and flags

- **The ios class contain a large number of member functions that would helps us to format the output in a number of ways.**
- **Width()**
- **Precision()**
- **Fill()**
- **Setf()**
- **Unsef()**

Width()

- **width() function to define the width of a field necessary for the output of an item. Since, it is a member function, we have to use an object to invoke it.**

Syntax

```
cout.width(w);
```

- **w- is the field width (number of column).**
- **The output will be printed in a field of w character wide at the right end of the field.**
- **width() can specify the field width only one item which immediately follows after that it will revert back to default.**

Example

```
#include<iostream>
using namespace std;
int main()
{
```

```
int item[4] = {10,8,12,15};
int const[4] = {75,100,60,99};

cout.width(5);
cout<< "ITEMS";
cout.width(8);
cout<<"COST";

cout.width(15);
cout<<"TOTAL VALUE"<<"\n";

int sum=0;
for(int i =0; i<4; i++)
{
cout.width(5);
cout<<item[i];

cout<<width(8);
cout<<cost[i];
int value = item[i] *cost[i];
cout.width(15);
cout<<value<<"\n";
sum =sum + value;
}
cout<<"\n" Grand Total =";
cout.width(2);
cout<<sum<<"\n";
return 0;
}
```

Output

ITEM	COST	TOTAL VALUE
10	75	750
8	100	800
12	60	720
15	99	1485

Precision()

- **By default, the floating numbers are printed with six digits after the decimal point.**
- **We can specify the number of digits to be displayed after the decimal point while printing the floating-point number.**
- **This can be done by using the precision() member function as follows:**

Syntax

```
cout.precision(d);
```

Example

```
#include<iostream>
#include<cmath>
using namespace std:
int main()
{
cout<<Precision set to 3 digits \n\n";
cout.precision(3);

cout.width(10);
cout<<"VALUE";
cout.width(15);
cout<<"SQRT_OF_VALUE"<<"\n";

for(int n =1; n<=5;n++)
{
 cout.width(8);
cout<<n;
cout.width(13);
cout<<sqrt(n) <<"\n";
}
cout <<"\n precision set to 5 digits \n\n";
cout.precision(5);
cout<<"sqrt(10) = " <<sqrt(10) << "\n\n";

cout.precision(0);
cout<< "sqrt(10) = " <<sqrt(10) <<"\n\n";
return 0;
}
```

Output

Precision set to 3 digit

VALUE	SQRT_OF_VALUE
1	1
2	1.41
3	1.73
4	2
5	2.24

Precision set to 5 digits

Sqrt(10) = 3.1623

Sqrt(10) = 3.162278 (default setting)

fill()

- **we have printing much large field width than required by the value.**
- **The unused position is filled with white space by default.**
- **We can use the fill() function to fill the unused position by any desired character.**

Syntax

```
cout.fill(ch);
```

Example

```
#include<iostream>
using namespace std;
int main()
{
cout.fill('<');
cout.precision(3);

for(int n=1; n<=6; n++)
{
cout.width(5);
cout<<n;
cout.width(10);
cout<< 1.0/float(n) << "\n";
if(n==3)
{
  cout.fill('>');
```

```
}
cout<<''\Padding changed \n\n";
cout.fill('#');
cout.width(15);
cout<<12.345678 <<"\n";
return 0;
}
```

Output:

<<<<1<<<<<<<<<<1
<<<<2<<<<<<<<<<0.5
<<<<3<<<<<<<<0.333
>>>>4>>>>>>>>>0.25
>>>>5>>>>>>>>>>0.2
>>>>6>>>>>>>>0.167

Padding changed
##########12.346

Formatting Flags setf()

- **When the width() is used the value and is printed right-justified in the field width created.**
- **But it is usual practice to print the text left-justified.**
- **Using setf() we can print text as left-justified and print with scientific notation.**
- **setf() – member function of the ios class.**

Syntax

```
cout.setf(arg1,arg2)
```

Example:

```
cout.fill('*');
cout.setf(ios::left,ios::adjustfield);
cout.width(15);
cout<<"TABLE 1"<<"\n"
```

Output

TABLE 1 ******

Manipulators

- **Manipulator are special functions that can be included in the I/O statements to alter the format parameters of a stream.**
- **To access these manipulators, the file iomanip should be included in the program.**
- **Setw()**
- **Setprecision()**
- **Setfill()**
- **Setioflags()**
- **Resetioflags()**

FILE HANDLING THROUGH C++ CLASSES

- Many real-life scenarios are there that handle a large number of data, and in such situations, you need to use some secondary storage to store the data.
- The data are stored in the secondary device using the concept of files.
- Files are the collection of related data stored in a particular area on the disk. Programs can be written to perform read and write operations on these files.

Working with files generally requires the following kinds of data communication methodologies:

- Data transfer between console units
- Data transfer between the program and the disk file

Standard file handling classes

1. **Ofstream:** This file handling class in C++ signifies the output file stream and is applied to create files for writing information to files
2. **Ifstream:** This file handling class in C++ signifies the input file stream and is applied for reading information from files
3. **fstream:** This file handling class in C++ signifies the file stream generally, and has the capabilities for representing both ofstream and ifstream

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files.

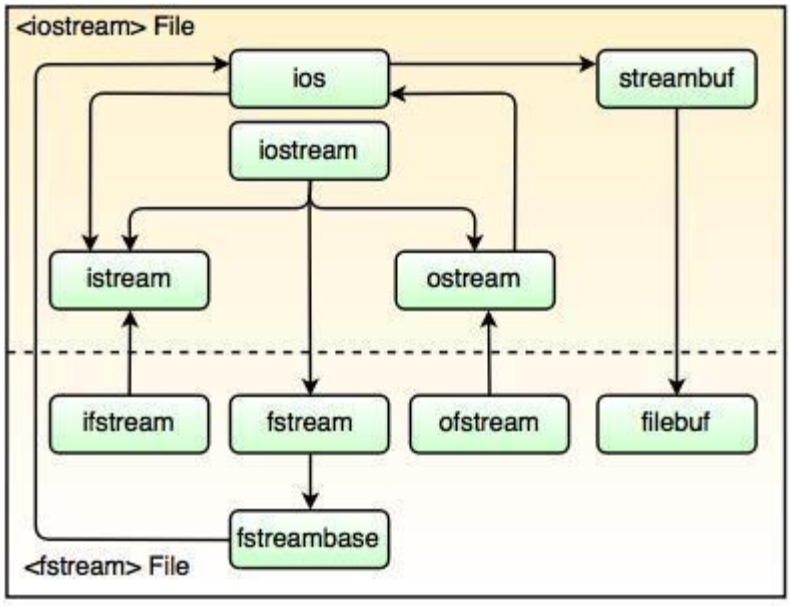


Fig. Stream Class Hierarchy

Opening and closing a file

If programmers want to use a disk file for storing data, they need to decide about the following things about the file and its intended use. These points that are to be noted are:

- A name for the file
- Data type and structure of the file
- Purpose (reading, writing data)
- Opening method
- Closing the file (after use)

Files can be opened in two ways. They are:

1. Using constructor function of the class
2. Using member function open of the class

OPENING A FILE

The first operation generally performed on an object of one of these classes to use a file is the procedure known as to opening a file. An open file is represented within a program by a stream and any input or output task performed on this stream will be applied to the physical file associated with it. The syntax of opening a file in C++ is:

```
open (filename, mode);
```

There are some mode flags used for file opening. These are:

- **ios::app**: append mode
- **ios::ate**: open a file in this mode for output and read/write controlling to the end of the file
- **ios::in**: open file in this mode for reading
- **ios::out**: open file in this mode for writing
- **ios::trunk**: when any file already exists, its contents will be truncated before file opening

CLOSING A FILE IN C++

When any C++ program terminates, it automatically flushes out all the streams releases all the allocated memory and closes all the opened files. But it is good to use the close() function to close the file related streams and it is a member of ifsream, ofstream and fstream objects.

The structure of using this function is:

```
void close();
```

General functions used for File handling

1. open(): To create a file
2. close(): To close an existing file
3. get(): to read a single character from the file
4. put(): to write a single character in the file
5. read(): to read data from a file
6. write(): to write data into a file

READING AND WRITING A FILE

- While doing C++ program, programmers write information to a file from the program using the stream insertion operator (<<) and reads information using the stream extraction operator (>>).

- The only difference is that for files programmers need to use an ofstream or fstream object instead of the cout object and ifstream or fstream object instead of the cin object.

Example:

```
#include <iostream>

#include <fstream.h>



void main () {

  ofstream file;

  file.open ("egone.txt");

  file << "Writing to a file in C++....";

  file.close();

  getch();

}
```

FILE POSITION POINTERS

- Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.
- The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction.
- The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

- The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

ERROR HANDLING DURING FILE OPERATION

- Sometimes during file operations, errors may also occur. For example, a file being opened for reading might not exist. Or a file name used for a new file may already exist. Or an attempt could be made to read past the end-of-file. Or such as invalid operation may be performed. There might not be enough space in the disk for storing data.
- To check for such errors and to ensure smooth processing, C++ file streams inherit 'stream-state' members from the ios class that store the information on the status of a file that is being currently used.
- The current state of the I/O system is held in an integer, in which the following flags are encoded

C++ ERROR HANDLING FUNCTIONS

There are several error handling functions supported by class ios that help you read and process the status recorded in a file stream.

Following table lists these error handling functions and their meaning :

Function	Meaning

int bad()	Returns a non-zero value if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operations.
int eof()	Returns non-zero (true value) if end-of-file is encountered while reading; otherwise returns zero (false value).
int fail()	Returns non-zero (true) when an input or output operation has failed.
int good()	Returns non-zero (true) if no error has occurred. This means, all the above functions are false. For example, if fin.good() is true, everything is okay with the stream named as fin and we can proceed to perform I/O operations. When it returns zero, no further operations can be carried out.
clear()	Resets the error state so that further operations can be attempted.

- The above functions can be summarized as eof() returns true if eofbit is set; bad() returns true if badbit is set.
- The fail() function returns true if failbit is set; the good() returns true there are no errors. Otherwise, they return false.
- These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby take the necessary corrective measures.

Example

```
#include<iostream.h>
#include<fstream.h>
#include<process.h>
#include<conio.h>
void main()
{
        clrscr();
        char fname[20];
        cout<<"Enter file name: ";
        cin.getline(fname, 20);
```

```
ifstream fin(fname, ios::in);
if(!fin)
{
        cout<<"Error in opening the file\n";
        cout<<"Press a key to exit...\n";
        getch();
        exit(1);
}
int val1, val2;
int res=0;
char op;
fin>>val1>>val2>>op;
switch(op)
{
        case '+':
                res = val1 + val2;
                cout<<"\n"<<val1<<" + "<<val2<<" = "<<res;
                break;
        case '-':
                res = val1 - val2;
                cout<<"\n"<<val1<<" - "<<val2<<" = "<<res;
                break;
        case '*':
                res = val1 * val2;
                cout<<"\n"<<val1<<" * "<<val2<<" = "<<res;
                break;
        case '/':
                if(val2==0)
                {
                        cout<<"\nDivide by Zero Error..!!\n";
                        cout<<"\nPress any key to exit...\n";
                        getch();
                        exit(2);
                }
                res = val1 / val2;
                cout<<"\n"<<val1<<" / "<<val2<<" = "<<res;
                break;
```

```
        }

        fin.close();

        cout<<"\n\nPress any key to exit...\n";
        getch();
}
```

TEMPLATES IN C++

- Templates are mostly implemented for crafting a family of classes or functions having similar features.
- For example, a class template for an array of the class would create an array having various data types such as float array and char array.
- Similarly, you can define a template for a function that helps you to create multiple versions of the same function for a specific purpose.
- A template can be considered as a type of macro; When a particular type of object is defined for use, then the template definition for that class is substituted with the required data type.
- A template can be considered as a formula or blueprints for generic class or function creations. This allows a function or class to work on different data types, without having to re-write them again for each.

Templates can be of two types in C++:

- Function templates
- Class templates

Function Templates

A function template defines a family of functions.

Syntax:

```
template < parameter-list > function-declaration



export template < parameter-list > function-declaration
```

- where *function-declaration* is the function name declared that becomes a template name and parameter-list is a non-empty comma-separated list of the template parameters.
- The general form of a function template is:

Syntax:

```
template <class type> ret-type func-name(parameter list)

{

   // body of function

}
```

- Function template plays a significant role, but it is neither a type by itself nor a function alone.
- It is not even an entity.
- They define a family of functions.
- No program gets generated from its source file which contains only template definitions.
- So, for showing a code, a template must have to be instantiated, i.e., the template arguments must have to be established so that the compiler can generate an actual function (or class, from a class template).

Syntax:

```
template returnType  nameOfTemplate <listOfArgument> (listOfParameter);
```

Example:

```
#include<iostream.h>

#include<conio.h>

template<class swap>



void swapp(swap &i, swap &j)
```

```
{
    swap t;
    t=i;
    i=j;
    j=t;
}

int main() {
    int e,f;
    char g,r;
    float x,y;

    cout<<"\n Please insert 2 Integer Values:"; cin>>e>>f;
    swapp(e,f);
    cout<<"\nInteger values after Swapping:";
    cout<<e<<"\t"<<f<<"\n\n";

    cout<<"\n Please insert 2 Character Values:"; cin>>g>>r;
    swapp(g,r);
    cout<<"\n Character Values after Swapping:";
```

```
    cout<<g<<"\t"<<r<<"\n\n";
    cout<<"\n please insert 2 Float Values:"; cin>>x>>y;
    swapp(x,y);
    cout<<"\n The resultant float values after swapping:";
    cout<<x<<"\t"<<y<<"\n\n";
}
```

Output:

```
Please insert 2 Integer Values: 12 10
Integer values after Swapping:1012

Please insert 2 Character Values: A B
Character Values after Swapping: A

Please insert 2 Float Values:1.1 2.1
The resultant float values after swapping:2.11.1
```

Class Template

A class template defines a family of classes.
Syntax:

```
template < parameter-list > class-declaration
```

- Where a class declaration is the class name which became the template name. Parameter - the list is a non-empty comma-separated list of the template parameters.
- A class template by itself is not a type, or an object, or any other entity.

- No code is generated from a source file that contains only template definitions.

Syntax:

```
template class name < argument-list >;       // Explicit instantiation definition


extern template class name < argument-list >;// Explicit instantiation declaration
```

Overloading of Template Function

- A template function may be overloaded wither by template function or by ordinary functions of its name.
- In such programming cases, the overloading resolution is accomplished as follows:

1. Call an ordinary function that has an exact match
2. A template function is called that could be created with an exact match
3. Try normal overloading resolution to ordinary functions and call the one that matches

Disadvantages of Using Template

1. Some compilers have poor support of template.
2. Many compilers lack clear instructions when they detect the error in the definition of the template.
3. Many compilers do not support nesting of templates.
4. When templates are used, all codes get exposed.
5.
6. The templates are in the header, in which the complete rebuild of all project pieces is required when the changes occur.

EXCEPTIONS HANDLING IN C++

- Exceptions allow a method to react to exceptional circumstances and errors (like runtime errors) within programs by transferring control to special functions called handlers.

- For catching exceptions, a portion of code is placed under exception inspection. Exception handling was not a part of the original C++.
- It is a new feature that ANSI C++ included in it. Now almost all C++ compilers support this feature.
- Exception handling technology offers a securely integrated approach to avoid the unusual predictable problems that arise while executing a program.

There are two types of exceptions:

1. Synchronous exceptions
2. Asynchronous exceptions
    - Errors such as: out of range index and overflow fall under the category of *synchronous* type exceptions.
    - Those errors that are caused by events beyond the control of the program are called *asynchronous* exceptions.
    - The main motive of the exceptional handling concept is to provide a means to detect and report an exception so that appropriate action can be taken.
    - This mechanism needs a separate error handling code that performs the following tasks:

- Find and hit the problem (exception)
- Inform that the error has occurred (throw exception)
- Receive the error information (Catch the exception)
- Take corrective actions (handle exception)

The error handling mechanism basically consists of two parts. These are:

1. To detect errors
2. To throw exceptions and then take appropriate actions

Exception handling in C++ is built on three keywords: *try*, *catch*, and *throw*.

- **try**
- **throw:** A program throws an exception when a problem is detected which is done using a keyword "throw".
- **catch:** A program catches an exception with an exception handler where programmers want to handle the anomaly. The keyword catch is used for catching exceptions.

The Catch blocks catching exceptions must immediately follow the try block that throws an exception.

Syntax:

```
try
{
   throw exception;
}

catch(type arg)
{
   //some code
}
```

- If the try block throws an exception then program control leaves the block and enters into the catch statement of the catch block.
- If the type of object thrown matches the argument type in the catch statement, the catch block is executed for handling the exception.
- Divided-by-zero is a common form of exception generally occurred in arithmetic based programs.

Example:

```
#include<iostream>

using namespace std;


int main()
```

```
{
   try {
      throw 6;
   }

   catch (int a) {
      cout << "An exception occurred!" << endl;
      cout << "Exception number is: " << a << endl;
   }
}
```

Example:

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

double division(int var1, int var2)
{
   if (var2 == 0) {
      throw "Division by Zero.";
   }
   return (var1 / var2);
}
```

```
int main()
{
   int a = 30;
   int b = 0;
   double d = 0;

   try {
      d = division(a, b);
      cout << d << endl;
   }
   catch (const char* error) {
      cout << error << endl;
   }

   return 0;
}
```

Output

```
Division by zero.
```

Advantages of Exception Handling

1. Programmers can deal with them at some level within the program
2. If an error can't be dealt with at one level, then it will automatically be shown at the next level, where it can be dealt with.

Unit V - Standard Template Library – Manipulating Strings – Object Oriented Systems Development

THE C++ STANDARD TEMPLATE LIBRARY (STL)

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.
- A working knowledge of template classes is a prerequisite for working with STL.

STL has four components

- Algorithms
- Containers
- Functions
- Iterators

Algorithms

- The header algorithm defines a collection of functions especially designed to be used on ranges of elements.
- They act on containers and provide means for various operations for the contents of the containers.

Algorithm

- Sorting
- Searching
- Important STL Algorithms
- Useful Array algorithms
- Partition Operations

Numeric

- valarray class

Containers

- Containers or container classes store objects and data.

- There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

Sequence Containers: implement data structures which can be accessed in a sequential manner.

- vector
- list
- deque
- arrays
- forward_list(Introduced in C++11)

Container Adaptors : provide a different interface for sequential containers.

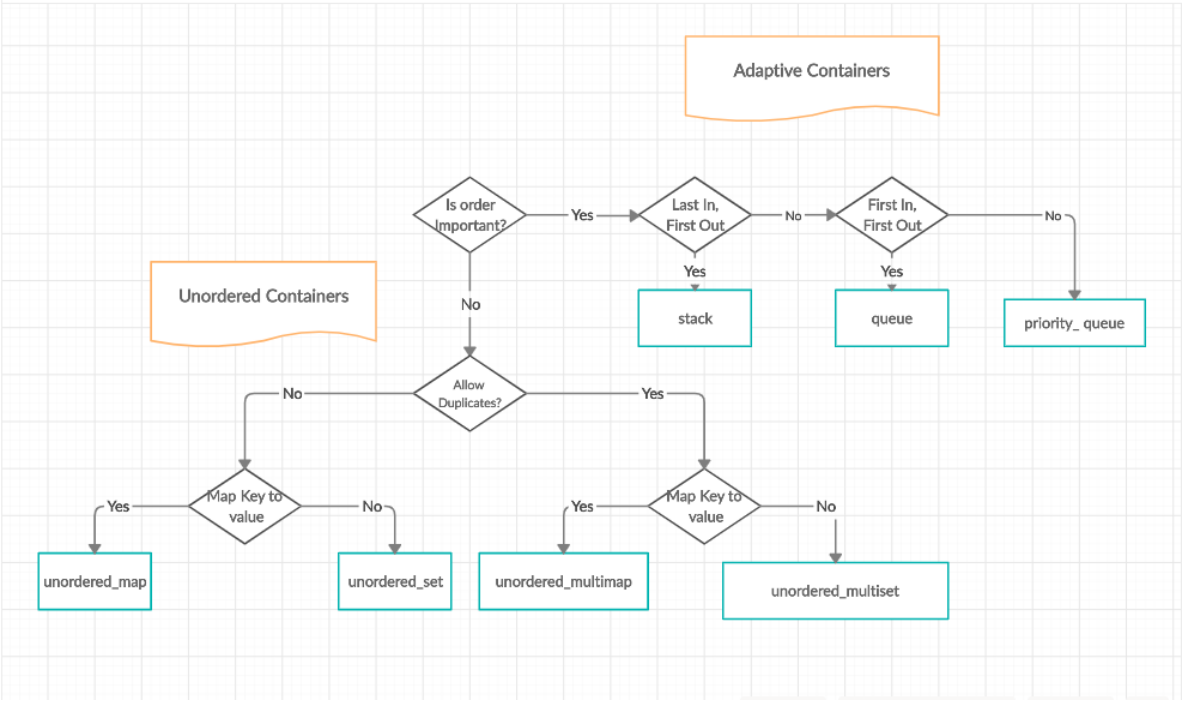
- queue
- priority_queue
- stack

Associative Containers : implement sorted data structures that can be quickly searched (O(log n) complexity).

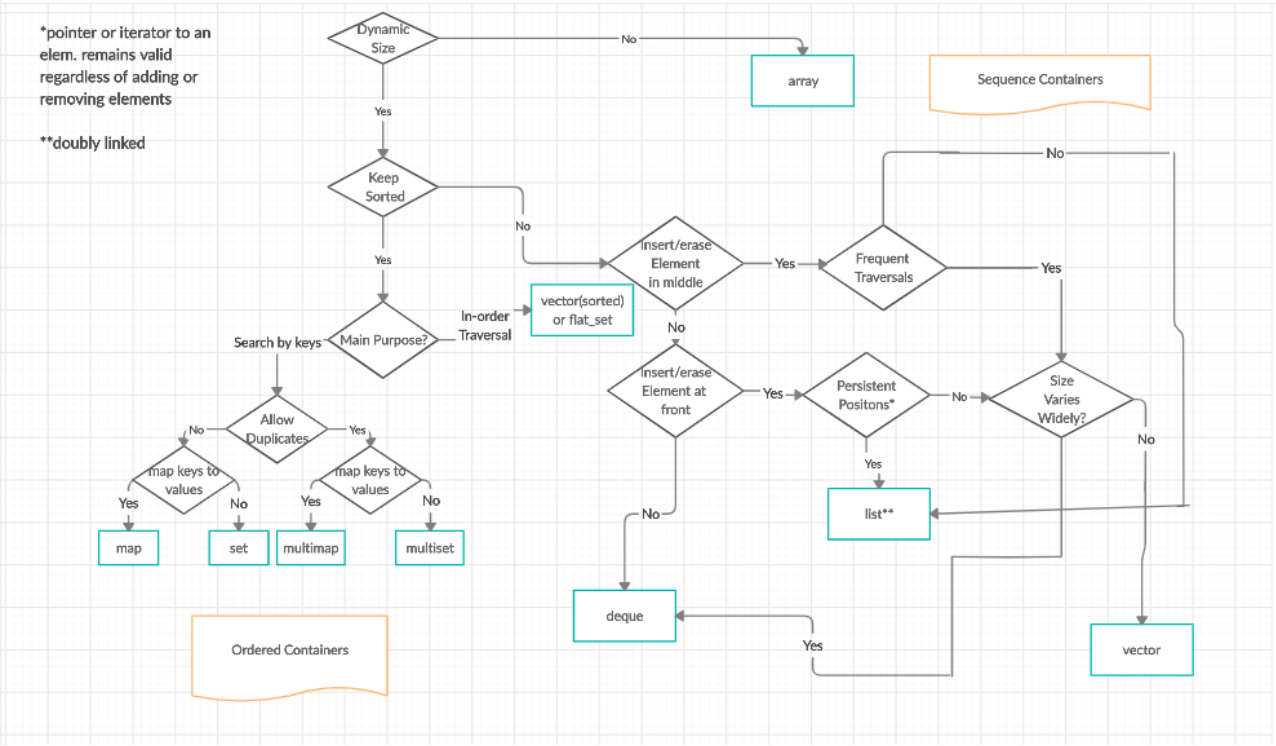
- set
- multiset
- map
- multimap

Unordered Associative Containers : implement unordered data structures that can be quickly searched

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap



Flowchart of Adaptive Containers and Unordered Containers



Flowchart of Sequence containers and ordered containers

Functions

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

- Functors

Iterators

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

- Iterators

Utility Library

Defined in header <utility>.

- pair

MANIPULATION OF STRING

C Style String

The C style string belongs to C language and continues to support in C++ also strings in C are the one-dimensional array of characters which gets terminated by \0 (null character).

This is how the strings in C are declared:

```
char ch[6] = {'H', 'e', 'l', 'l', 'o', '

char ch[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

'};
```

Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the \0 at the end of the string when it initializes the array.

String Class in C++

The string class is huge and includes many constructors, member functions, and operators.

- Creating string objects
- Reading string objects from keyboard
- Displaying string objects to the screen
- Finding a substring from a string
- Modifying string
- Adding objects of string
- Comparing strings

- Accessing characters of a string
- Obtaining the size or length of a string, etc...

Manipulate Null-terminated strings

- strcpy(str1, str2): Copies string str2 into string str1.
- strcat(str1, str2): Concatenates string str2 onto the end of string str1.
- strlen(str1): Returns the length of string str1.
- strcmp(str1, str2): Returns 0 if str1 and str2 are the same; less than 0 if str1<str2; greater than 0 if str1>str2.
- strchr(str1, ch): Returns a pointer to the first occurrence of character ch in string str1.
- strstr(str1, str2): Returns a pointer to the first occurrence of string str2 in string str1.

Important functions supported by String Class

- append(): This function appends a part of a string to another string
- assign():This function assigns a partial string
- at(): This function obtains the character stored at a specified location
- begin(): This function returns a reference to the start of the string
- capacity(): This function gives the total element that can be stored
- compare(): This function compares a string against the invoking string
- empty(): This function returns true if the string is empty
- end(): This function returns a reference to the end of the string
- erase(): This function removes character as specified
- find(): This function searches for the occurrence of a specified substring
- length(): It gives the size of a string or the number of elements of a string
- swap(): This function swaps the given string with the invoking one

Important Constructors obtained by String Class

- String(): This constructor is used for creating an empty string
- String(const char *str): This constructor is used for creating string objects from a null-terminated string
- String(const string *str): This constructor is used for creating a string object from another string object

Operators used for String Objects

1. =: assignment
2. +: concatenation
3. ==: Equality
4. !=: Inequality
5. <: Less than
6. <=: Less than or equal
7. >: Greater than
8. >=: Greater than or equal
9. []: Subscription
10. <<: Output
11. >>: Input

Example

```
#include <iostream>
#include <cstring>

using namespace std;

int main () {

   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;

   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;

   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;

   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```

Output

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

Example

```
#include <iostream>
#include <string>

using namespace std;

int main () {

   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // total length of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

Output

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

- As seen in the above code, we can get the length of the string by size() as well as length() but length() is preferred for strings.
- We can concat a string to another string by += or by append(), but += is slightly slower than append() because each time + is called a new string (creation of new buffer) is made which is returned that is a bit overhead in case of many append operation.