C++ IMP QUESTION

What is Constructors? Type of Constructors.

Constructor:-

1. constructors are the special member function of the class which are used to initialize the objects of that class
2. The constructor of class is automatically called immediately after the creation of the object.
3. Name of the constructor should be exactly same as that of name of the class.
4. constructor is called only once in a lifetime of object when object is created.
5. constructors can be overloaded
6. constructors does not return any value so constructor have no return type.
7. constructor does not return even void as return type.

The various types of Constructor are as follows:-

1. **Default Constructor:-** Default Constructor is also called as Empty Constructor which has no arguments and It is Automatically called when we creates the object of class but Remember name of Constructor is same as name of class and Constructor never declared with the help of Return Type. Means we cant Declare a Constructor with the help of void Return Type.

- If the programmer does not specify the constructor in the program then compiler provides the default constructor.
- In C++ we can overload the default compiler generated constructor
- In both cases (user created default constructor or default constructor generated by compiler), the default constructor is always parameterless.

Example of Default Constructor

Let us take the example of class Marks which contains the marks of two subjects Maths and Science.

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

class Marks
{
public:
  int maths;
  int science;

  //Default Constructor
  Marks() {
    maths=0;
    science=0;
  }

  display() {
    cout << "Maths :  " << maths <<endl;
    cout << "Science :" << science << endl;
  }
};

int main() {
 //invoke Default Constructor
```

```
  Marks m;
  m.display();
  return 0;
}
```

Output :

```
Maths :  0
Science : 0
```

2. **Parameterized Constructor :-** This is Another type of Constructor which has some Arguments and same name as class name but it uses some Arguments So For this We have to create object of Class by passing some Arguments at the time of creating object with the name of class.

Syntax:

```
class_name(Argument_List) {
-----
-----
}
```

Example of Parametrized Constructor:

Let us take the example of class 'Marks' which contains the marks of two subjects Maths and Science.

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

```
class Marks
{
public:
  int maths;
  int science;

  //Parametrized Constructor
  Marks(int mark1,int mark2) {
    maths = mark1;
    science = mark2;
  }

  display() {
    cout << "Maths :  " << maths <<endl;
    cout << "Science :" << science << endl;
  }
};

int main() {
 //invoke Parametrized Constructor
 Marks m(90,85);
 m.display();
 return 0;
}
```

3. **Copy Constructor:-** This is also Another type of Constructor. In this Constructor we pass the value of one object of class into the Another Object of Same Class. As name Suggests you Copy, means Copy the values of one Object into the another Object of Class .This is used for Copying the

values of class object into an another object of class So we call them as Copy Constructor.

1. All member values of one object can be assigned to the other object using copy constructor.
2. For copying the object values, both objects must belong to same class.

Syntax:

```
class_Name (const class_Name &obj) {
  // body of constructor
}
```


Example of Copy Constructor

Let us take the example of class 'Marks' which contains the marks of two subjects Maths and Science.

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

class marks
{
public:
   int maths;
   int science;

   //Default Constructor
   marks(){
```

```
    maths=0;
    science=0;
  }

  //Copy Constructor
  marks(const marks &obj){
    maths=obj.maths;
    science=obj.science;
  }

  display(){
    cout<<"Maths :   " << maths
    cout<<"Science : " << science;
  }
};

int main(){
  marks m1;

  /*default constructor gets called
      for initialization of m1  */
  marks m2(const marks &m1);

  //invoke Copy Constructor
  m2.display();

  return 0;
}
```

Output

```
Maths :  0
Science : 0
```

Destructor

1. Destructors are special member functions of the class required to free the memory of the object whenever it goes out of scope.
2. Destructors are parameterless functions.
3. Name of the Destructor should be exactly same as that of name of the class. But preceded by '~' (tilde).
4. Destructors does not have any return type. Not even void.
5. The Destructor of class is automatically called when object goes out of scope.

C++ Destructor programs

C++ Destructor Program #1 : Simple Example

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

class Marks
{
public:
```

```
  int maths;
  int science;

  //constructor
  Marks() {
    cout << "Inside Constructor"<<endl;
    cout << "C++ Object created"<<endl;
  }

  //Destructor
  ~Marks() {
    cout << "Inside Destructor"<<endl;
    cout << "C++ Object destructed"<<endl;
  }
};

int main( )
{
  Marks m1;
  Marks m2;
  return 0;
}
```

Output

```
Inside Constructor
C++ Object created
Inside Constructor
```

```
C++ Object created


Inside Destructor
C++ Object destructed
Inside Destructor
C++ Object destructed
```

Explanation :

You can see destructor gets called just before the return statement of main function. You can see destructor code below –

```
~Marks() {
  cout << "Inside Destructor"<<endl;
  cout << "C++ Object destructed"<<endl;
}
```

C++ Destructor always have same name as that of constructor but it just identified by tilde (~) symbol before constructor name.

STL

STL (standard template library). It is a set of C++ template classes that provide generic classes and function that can be used to implement data structures and algorithms .STL is mainly composed of:

1. Algorithms
2. Containers
3. Iterators

STL

STL provides numerous containers and algorithms which are very useful in completive programming, for example you can very easily define a linked list in a single statement by using list container of container library in STL, saving your time and effort.

STL is a generic library, i.e. a same container or algorithm can be operated on any data types, you don't have to define the same algorithm for different type of elements.

For example, sort algorithm will sort the elements in the given range irrespective of their data type, we don't have to implement different sort algorithm for different datatypes.

Algorithms in STL

STL provide number of algorithms that can be used of any container, irrespective of their type. Algorithms library contains built in functions that performs complex algorithms on the data structures.

For example: one can reverse a range with `reverse()` function, sort a range with `sort()` function, search in a range with `binary_search()` and so on.

Algorithm library provides abstraction, i.e you don't necessarily need to know how the the algorithm works.

Containers in STL

Container library in STL provide containers that are used to create data structures like arrays, linked list, trees etc.

These container are generic, they can hold elements of any data types, for example: **vector** can be used for creating dynamic arrays of char, integer, float and other types.

Iterators in STL

Iterators in STL are used to point to the containers. Iterators actually acts as a bridge between containers and algorithms.

For example: `sort()` algorithm have two parameters, starting iterator and ending iterator, now sort() compare the elements pointed by each of these iterators and arrange them in sorted order, thus it does not matter what is the type of the container and same sort() can be used on different types of containers.

Inheritance & types of inheritance:

Inheritance:

The process of obtaining the data members & data from one class to another class is called inheritance.

It is one fundamental features of object-oriented programming.

For eg: A child acquires property of parents.

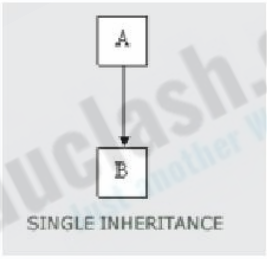
Syntax:

```
class DerivedClass : accessSpecifier BaseClass
```

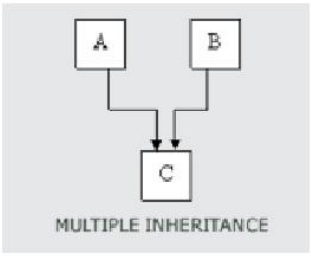
Access specifier can be public, protected and private. The default access specifier is **private.** Access specifiers affect accessibility of data members of base class from the derived class. In addition, it determines the accessibility of data members of base class outside the derived class.

Types of inheritance:

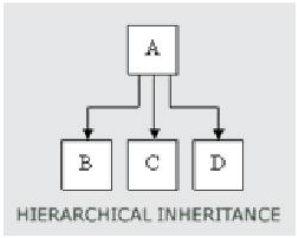
1. Single inheritance: It is inheritance hierarchy where in one derive class inherits from one base class.



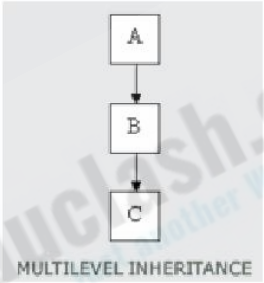
2. Multiple inheritance: It is inheritance hierarchy where in one derived class inherits from multiple base class(es)



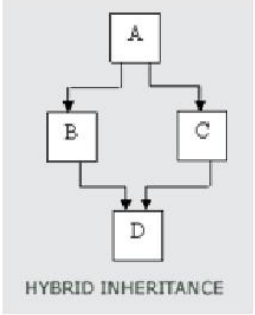
3. Hierarchical Inheritance: It is the inheritance hierarchy where in multiple subclasses inerit from one base class.



4. Multilevel inheritance: It is the inheritance hierarchy where in subclass acts as a base class for other classes.



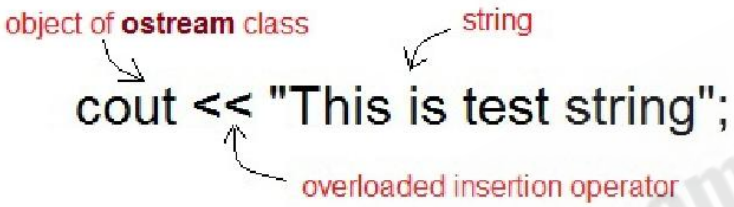
5. Hybrid inheritance: The inheritance hierarchy that reflects any legal combination of other four types of inheritance



Operator Overloading

Operator Overloading:

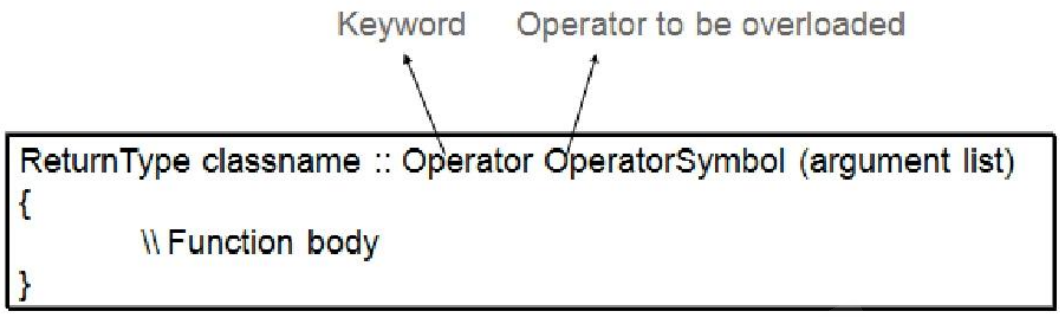
Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various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



Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Operator Overloading Syntax



Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be:

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Exception Handling

Exception Handling:

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  // protected code
} catch( ExceptionName e1 )
{
  // catch block
} catch( ExceptionName e2 )
{
  // catch block
} catch( ExceptionName eN )
{
  // catch block
}
```

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b) {
  if( b == 0 ) {
     throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
  // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown

in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try {
   // protected code
} catch(...) {
   // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

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

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
      cerr << msg << endl;
```

```
   }
   return 0;
}
```

Because we are raising an exception of type **const char*** , so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

```
Division by zero condition!
```

Function overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types:

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

class printData {
   public:
      void print(int i) {
         cout << "Printing int: " << i << endl;
      }

      void print(double  f) {
         cout << "Printing float: " << f << endl;
      }

      void print(char* c) {
```

```
      cout << "Printing character: " << c << endl;
   }
};

int main(void) {
  printData pd;

  // Call print to print integer
  pd.print(5);

  // Call print to print float
  pd.print(500.263);

  // Call print to print character
  pd.print("Hello C++");

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Friend Function

One of the important concepts of OOP is data hiding, i.e., a nonmember function cannot access an object's private or protected data.

But, sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member functions.

This is done using a friend function or/and a friend class.

friend Function in C++

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.

The complier knows a given function is a friend function by the use of the keyword **friend**.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword **friend**.

Declaration of friend function in C++

```
class class_name

{

   ... .. ...

   friend return_type function_name(argument/s);

   ... .. ...

}
```

Now, you can define the friend function as a normal function to access the data of the class. No **friend** keyword is used in the definition.

```
class className
```

```
{
    ... .. ...
    friend return_type functionName(argument/s);
    ... .. ...
}


return_type functionName(argument/s)
{
    ... .. ...
    // Private and protected data of className can be accessed from
    // this function because it is a friend function of className.
    ... .. ...
}
```

Example 1: Working of friend Function

```
/* C++ program to demonstrate the working of friend function.*/
#include <iostream>
using namespace std;
```

```
class Distance
{
   private:
      int meter;
   public:
      Distance(): meter(0) { }
      //friend function
      friend int addFive(Distance);
};

// friend function definition
int addFive(Distance d)
{
   //accessing private data from non-member function
   d.meter += 5;
   return d.meter;
}

int main()
{
   Distance D;
   cout<<"Distance: "<< addFive(D);
   return 0;
}
```

Output

```
Distance: 5
```

Abstract class and Pure virtual Function

The goal of object-oriented programming is to divide a complex problem into small sets. This helps understand and work with problem in an efficient way.

Sometimes, it's desirable to use inheritance just for the case of better visualization of the problem.

In C++, you can create an abstract class that cannot be instantiated (you cannot create object of that class). However, you can derive a class from it and instantiate object of the derived class.

Abstract classes are the base class which cannot be instantiated.

A class containing pure virtual function is known as abstract class.

Pure Virtual Function

A virtual function whose declaration ends with **= 0** is called a pure virtual function. For example,

```
class Weapon

{

   public:

     virtual void features() = 0;

};
```

Here, the pure virtual function is

```
virtual void features() = 0
```

And, the class **Weapon** is an abstract class.

Example: Abstract Class and Pure Virtual Function

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

// Abstract class
class Shape
{
   protected:
      float l;
   public:
      void getData()
      {
         cin >> l;
      }

      // virtual Function
      virtual float calculateArea() = 0;
};

class Square : public Shape
{
   public:
      float calculateArea()
      {   return l*l;  }
};
```

```
class Circle : public Shape
{
   public:
      float calculateArea()
      { return 3.14*l*l; }
};

int main()
{
   Square s;
   Circle c;

   cout << "Enter length to calculate the area of a square: ";
   s.getData();
   cout<<"Area of square: " << s.calculateArea();
   cout<<"\nEnter radius to calculate the area of a circle: ";
   c.getData();
   cout << "Area of circle: " << c.calculateArea();

   return 0;
}
```

Output

```
Enter length to calculate the area of a square: 4

Area of square: 16

Enter radius to calculate the area of a circle: 5
```

```
Area of circle: 78.5
```

In this program, pure virtual function **virtual float area() = 0;** is defined inside the **Shape** class.

One important thing to note is that, you should override the pure virtual function of the base class in the derived class. If you fail the override it, the derived class will become an abstract class as well.

Pass by value and Pass by Reference

We have seen as to how to pass values to a function through arguments. Actually there are two ways to pass values to a function through arguments. These two methods are explained below with examples.

Pass By Value:

Example:

```
void check (int x)
{

//body of function
}
int main ( )
{
int b = 10;
check (b);
}
```

In this function, 'x' is a parameter and 'b' (which is the value to be passed) is the argument. In this case the value of 'b' (the argument) is copied in 'x' (the parameter). Hence the parameter is actually a copy of the argument.

The function will operate only on the copy and not on the original argument. This method is known as PASS BY VALUE. So far we have dealt only with pass by value (except when passing arrays to functions).

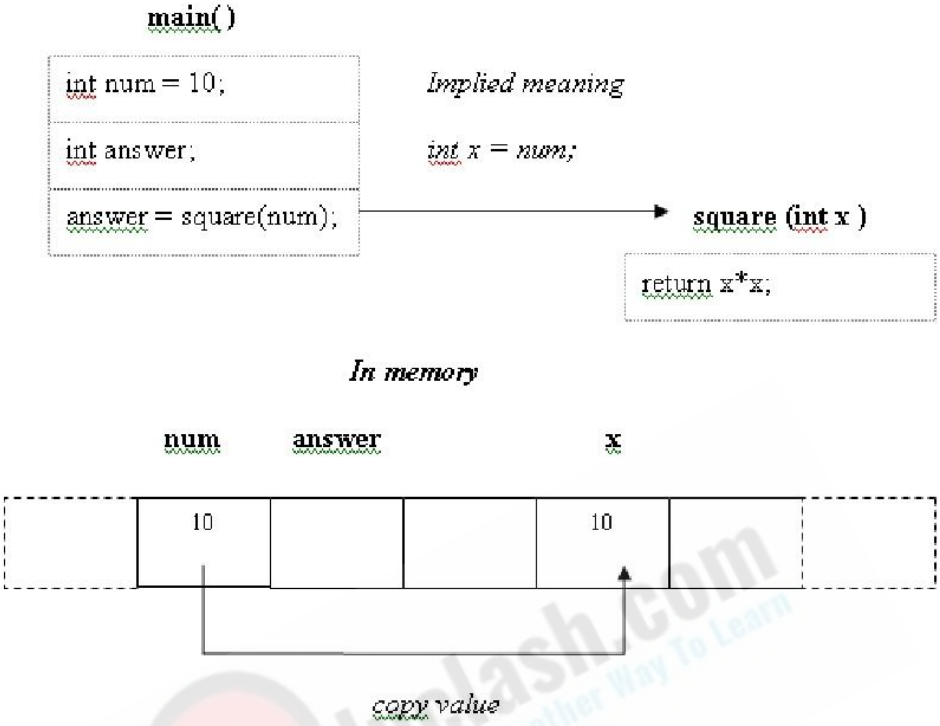
```
// Pass by value illustration

# include <iostream.h>

int square (int x)
{
return x*x;
}

int main ( )
{
int num = 10;
int answer;
answer = square(num);
cout<<"Answer is "<<answer; // answer is 100
cout<<" Value of a is "<<num; // num will be 10
return 0;
}
```

You can see that the value of 'num' is unchanged. The function 'square' works only on the parameter (i.e. on x). It does not work on the original variable that was passed (i.e it doesn't work on 'num').



The diagram makes it quite clear as to what happens when we call square(num). The following initialization takes place:

```
int x = num;
```

and the function square( ) operates only on a copy of num.

Pass By Reference

In pass by reference method, the function will operate on the original variable itself. It doesn't work on a copy of the argument but works on the argument itself. Consider the same square function example:

```
// Illustration of pass by reference
```

```
#include <iostream.h>

void square (int *x)
{
*x = (*x) * (*x);
}

int main ( )
{
int num = 10;
square(&num);
cout<<" Value of num is "<<num; // Value of num is 100
return 0;
}
```

As you can see the result will be that the value of a is 100. The idea is simple: the argument passed is the address of the variable 'num'. The parameter of 'square' function is a pointer pointing to type integer. The address of 'num' is assigned to this pointer. You can analyze it as follows: &num is passed to int *x, therefore it is the same as:

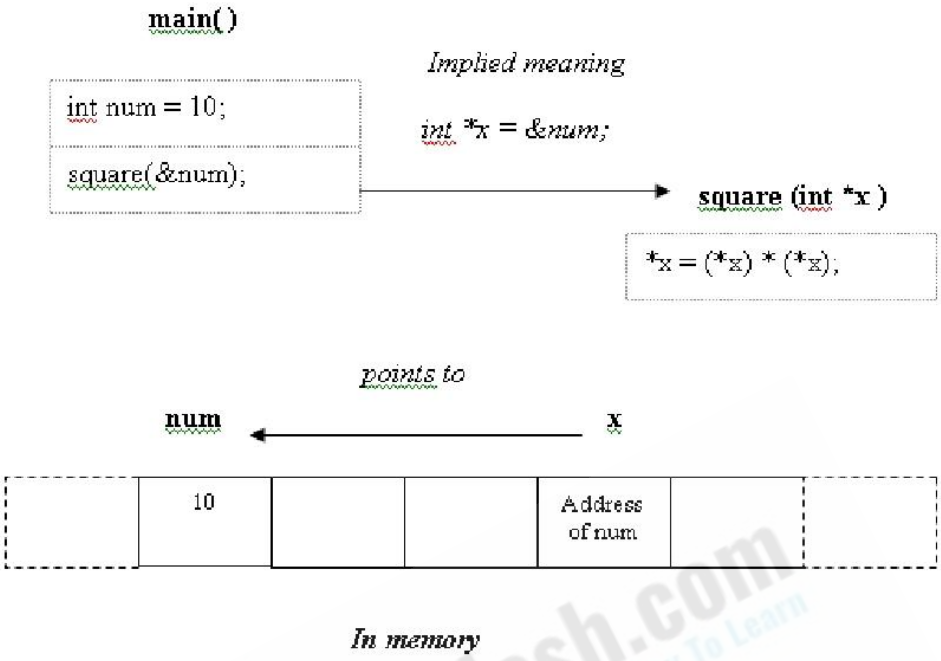
```
int *x = &num;
```

This means that 'x' is a pointer to an integer and has the address of the variable num.

Within the function we have:

```
*x = (*x) * (*x);
```

* when used before a pointer will give the value stored at that particular address. Hence we find the product of 'num' and store it in 'num' itself. i.e. the value of 100 is stored in the address of 'num' instead of 10 which was originally present there. The diagram below illustrates the difference between pass by value and pass by reference. Now when we dereference 'x' we are actually manipulating the value stored in 'num'.



This is the pass-by-reference method which was used in C. In C++ there is a different approach. Of course you can use the above method, but C++ has its own special way.

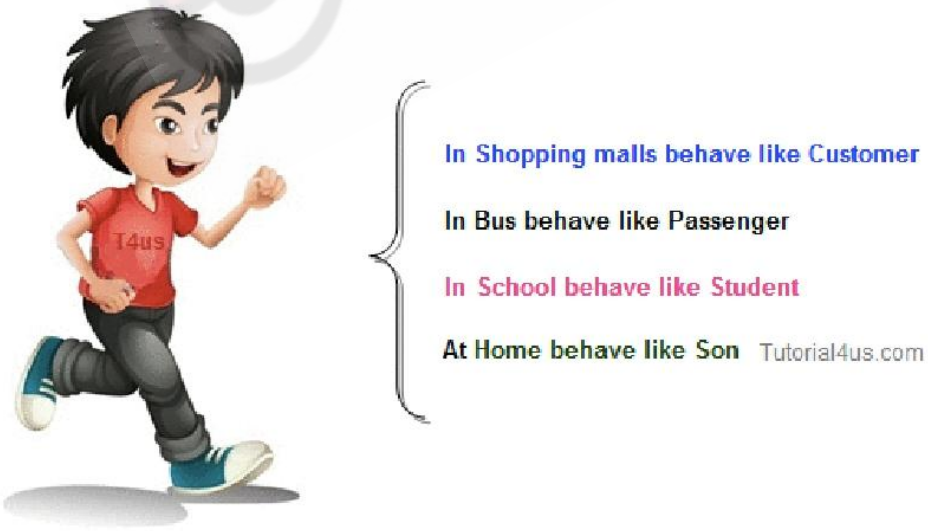
Polymorphism in C++

The process of representing one Form in multiple forms is known as **Polymorphism**. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.

Polymorphism is derived from 2 greek words: **poly** and morphs. The word "poly" means many and **morphs** means forms. So polymorphism means many forms.

Real life example of Polymorphism in C++

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.



Type of polymorphism

- Compile time polymorphism
- Run time polymorphism

Compile time polymorphism

In C++ programming you can achieve compile time polymorphism in two way, which is given below;

- Method overloading
- Method overriding

Method Overloading in C++

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**. In below example method "sum()" is present in Addition class with same name but with different signature or arguments.

Example of Method Overloading in C++

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

class Addition
{
public:
void sum(int a, int b)
{
cout<<a+b;
}
void sum(int a, int b, int c)
{
cout<<a+b+c;
}
};
```

```
void main()
{
clrscr();
Addition obj;
obj.sum(10, 20);
cout<<endl;
obj.sum(10, 20, 30);
}
```

Output

```
30

60
```

Method Overriding in C++

Define any method in both base class and derived class with same name, same parameters or signature, this concept is known as **method overriding**. In below example same method "show()" is present in both base and derived class with same name and signature.

Example of Method Overriding in C++

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

class Base
{
 public:
 void show()
 {
  cout<<"Base class";
 }
};
```

```
class Derived:public Base
{
 public:
 void show()
 {
  cout<<"Derived Class";
 }
}

int mian()
{
 Base b;       //Base class object
 Derived d;  //Derived class object
 b.show();     //Early Binding Ocuurs
 d.show();
getch();
}
```

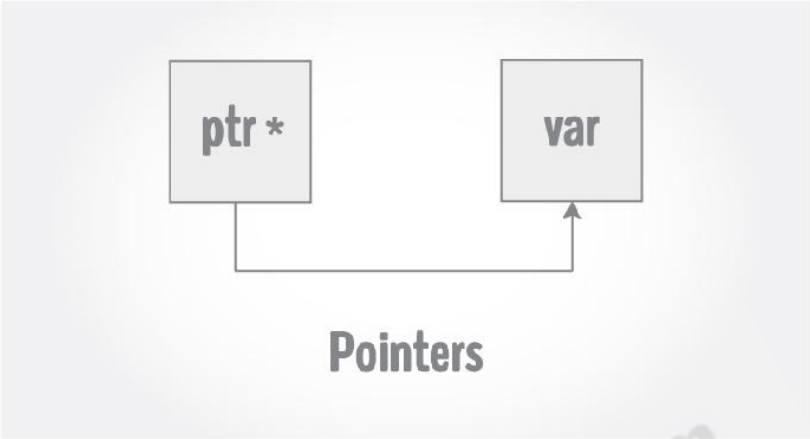
Output

```
Base class
Derived Class
```

Run time polymorphism

In C++ Run time polymorphism can be achieve by using virtual function.

C++ Pointers:



Pointers are powerful features of C++ that differentiates it from other programming languages like Java and Python.

Pointers are used in C++ program to access the memory and manipulate the address.

Address in C++

To understand pointers, you should first know how data is stored on the computer.

Each variable you create in your program is assigned a location in the computer's memory. The value the variable stores is actually stored in the location assigned.

To know where the data is stored, C++ has an & operator. The & (reference) operator gives you the address occupied by a variable.

If **var** is a variable then, **&var** gives the address of that variable.

Example 1: Address in C++

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

int main()
{
    int var1 = 3;
    int var2 = 24;
    int var3 = 17;
    cout << &var1 << endl;
    cout << &var2 << endl;
    cout << &var3 << endl;
}
```

Output

```
0x7fff5fbff8ac

0x7fff5fbff8a8

0x7fff5fbff8a4
```

Note: You may not get the same result on your system.

The **0x** in the beginning represents the address is in hexadecimal form.

Notice that first address differs from second by 4-bytes and second address differs from third by 4-bytes.

This is because the size of integer (variable of type **int**) is 4 bytes in 64-bit system.

Pointers Variables

C++ gives you the power to manipulate the data in the computer's memory directly. You can assign and de-assign any space in the memory as you wish. This is done using Pointer variables.

Pointers variables are variables that points to a specific address in the memory pointed by another variable.

How to declare a pointer?

```
int *p;

      OR,

int* p;
```

The statement above defines a pointer variable **p**. It holds the memory address

The asterisk is a dereference operator which means **pointer to.**

Here, pointer **p** is a **pointer to int**, i.e., it is pointing to an integer value in the memory address.

Reference operator (&) and Deference operator (*)

Reference operator (&) as discussed above gives the address of a variable.

To get the value stored in the memory address, we use the dereference operator (*).

For example: If a **number** variable is stored in the memory address **0x123**, and it contains a value **5**.

The **reference (&)** operator gives the value **0x123**, while the **dereference (*)** operator gives the value **5**.

Note: The (*) sign used in the declaration of C++ pointer is not the dereference pointer. It is just a similar notation that creates a pointer.

Example 2: C++ Pointers

C++ Program to demonstrate the working of pointer.

```
#include <iostream>
using namespace std;
int main() {
    int *pc, c;

    c = 5;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;

    pc = &c;    // Pointer pc holds the memory address of variable c
    cout << "Address that pointer pc holds (pc): "<< pc << endl;
```

```
    cout << "Content of the address pointer pc holds (*pc): " << *pc <<
endl << endl;


    c = 11;    // The content inside memory address &c is changed from 5
to 11.
    cout << "Address pointer pc holds (pc): " << pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc <<
endl << endl;


    *pc = 2;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;


    return 0;
}
```

Output

```
Address of c (&c): 0x7fff5fbff80c
Value of c (c): 5

Address that pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 5

Address pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 11

Address of c (&c): 0x7fff5fbff80c
Value of c (c): 2
```

Multiple Inheritance in C++:

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.
The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B's constructor is called before A's constructor.

```
# include< iostream >

using namespace std;


class A

{

public:

  A()  {  cout << "A's constructor called" << endl; }

} ;


class B

{

public:

  B()  {  cout << "B's constructor called" << endl; }

} ;


class C: public B, public A  // Note the order

{

public:

  C()  {  cout << "C's constructor called" << endl; }

} ;


int main()
```

```
{
    C c;
    return 0;
}
```

Run on IDE

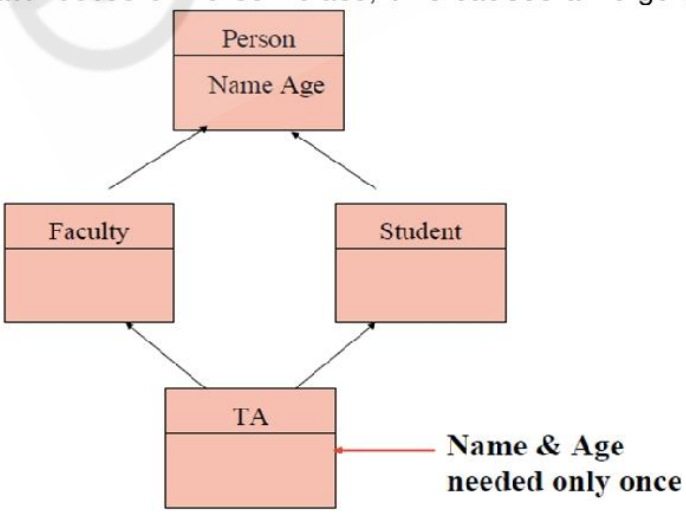
Output:

```
B's constructor called
A's constructor called
C's constructor called
```

The destructors are called in reverse order of constructors.

The diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



For example, consider the following program.

```
# include< iostream >
```

```
using namespace std;
class Person {
  // Data members of person
public:
   Person(int x) { cout << "Person::Person(int ) called" << endl;  }
};

class Faculty : public Person {
  // data members of Faculty
public:
   Faculty(int x):Person(x)  {
     cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : public Person {
  // data members of Student
public:
   Student(int x):Person(x) {
      cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x)  {
      cout<<"TA::TA(int ) called"<< endl;
```

```
    }
};


int main() {
    TA ta1(30);
}
```

Run on IDE

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person', this causes ambiguities. *The solution to this problem is 'virtual' keyword*. We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class. For example, consider the following program.

Access specifiers:

C++ offers possibility to control access to class members and functions by using access specifiers. Access specifiers are used to protect data from misuse.

In the Person class from the previous topic we used only **public** access specifiers for all data members:

Types of access specifiers in C++

1. public
2. private
3. protected

Public Specifier:

Public class members and functions can be used from outside of a class by any function or other classes. You can access public data members or function directly by using **dot** operator (.) or (arrow operator-> with pointers).

Protected Specifier:

Protected class members and functions can be used inside its class. Protected members and functions cannot be accessed from other classes directly. Additionally **protected** access specifier allows friend functions and classes to access these data members and functions. **Protected** data

members and functions can be used by the class derived from this class. More information about access modifiers and inheritance can be found in **C++ Inheritance**

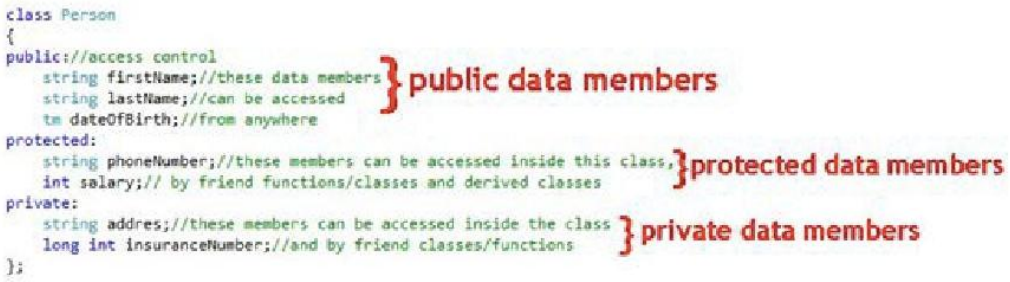
Private Specifier:

Private class members and functions can be used only inside of class and by friend functions and classes.

We can modify **Person** class by adding data members and function with different access specifiers:

```
class Person
{
public:                    //access control
      string firstName;        //these data members
      string lastName;         //can be accessed
      tm dateOfBirth;          //from anywhere
protected:
      string phoneNumber;   //these members can be accessed inside this class,
      int salary;              // by friend functions/classes and derived classes
private:
      string addres;           //these members can be accessed inside the class
      long int insuranceNumber;//and by friend classes/functions
};
```

Access specifier affects all the members and functions until the next access specifier:



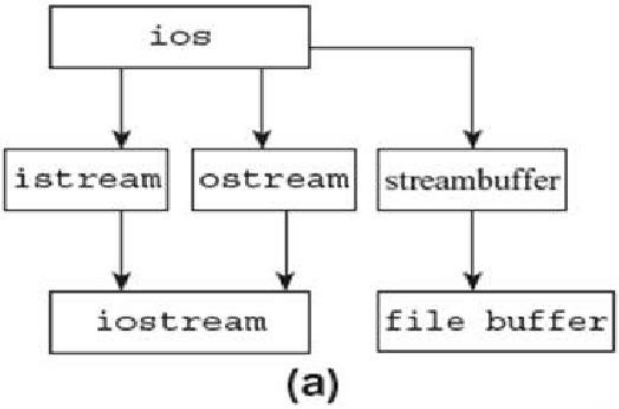
For classes, default access specifier is **private.** The default access specifier for unions and structs is public.

Stream Classes:

C++ streams are based on class and object theory. C++ has number of classes that work with console and file operations. These classes are known as stream classes. Figure shows the stream classes. All these classes are declared in the header file iostream.h. The file iostream.h must be included in the program, if we are using the functions of these classes.

As described in Figure (a) the classes istream and ostream are derived classes of base class ios. The ios class contains member variable object streambuf. The streambuf places the buffer. The member function of streambuf class handles the buffer by providing the facilities to flush clear and pour the buffer. The class iostream is derived from the classes istream and ostream by using multiple inheritance. The ios class is a virtual class and it is present to avoid ambiguity that frequently appears in multiple inheritance. Chapter 11 describes multiple inheritance and virtual classes.

The ios class has an ability to handle formatted and unformatted I/O operations. The istream class gives support for both formatted and unformatted data. The ostream classes handle the formatting of output data. The iostream contains functions of both istream and ostream classes. The classes istream_withassign, ostream_withassign, and iostream_withassign append appropriate assignment operators as shown in Figure (b) Table describes functions/contents of C++ stream classes.



(a)

Hierarchy of stream class

Table: Functions/Contents of C++ Stream Classes	
Class	**Function/Contents**
ios	(1) It is an input and output stream class. (2) It is used to implement a buffer, i.e. it is pointer to a buffer streambuf. (3) ios maintains the information on the state of streambuf, i.e. good, bad, eof, etc.
istream	(1) istream provides formatted input. (2) It is used to handle formatted as well as unformatted conversion of character from a streambuf. (3) The properties of ios are inherited in istream class. (4) The instance of class does not carry out the actual input. (5) istream declares functions such as peek(), tellg(), seekg(), getline(), read(), etc. (6) istream class overloads the '>>' operator.
ostream	(1) It is used for general-purpose output. (2) It is used to declare the output functions such as tellp(), put(), write(), seekp(), etc. (3) It is the parent of all output stream. (4) ostream overloads the '<<' operator.
iostream	(1) It is used to handle both input and output streams.
istream_withassign	(1) It is derived from istream. (2) It is used for cin input.
iostream_withassign	(1) It is a bidirectional stream.