OOPS IN C++ (PART-1)

PLACEMENT PREPARATION [EXCLUSIVE NOTES]

SAVE AND SHARE

Curated By- HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

TOP QUESTIONS THAT ARE COVERED: -

PART-1: -

- Encapsulation in C++
- Polymorphism in C++
- Abstraction in C++
- Inheritance in C++
- Inline Functions in C++
- Operator Overloading in C++
- Function Overloading in C++
- Shallow Copy and Deep Copy in C++
- Application of OOPs in C++

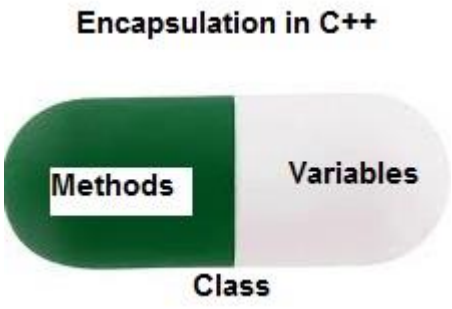
PART-2: -

- Object Oriented Programming in C++
- What is a class?
- What is an object?
- What is encapsulation?
- What is Polymorphism?
- What is Compile time Polymorphism and how is it different from Runtime Polymorphism?
- How does C++ support Polymorphism?
- What is meant by Inheritance?
- What are the various types of inheritance?
- Why multiple inheritance is not supported in JAVA?
- What is Abstraction?
- How much memory does a class occupy?
- Is it always necessary to create objects from class in C++?
- What is a constructor?
- What are the various types of constructors in C++?

- **What is a copy constructor?**
- **What is a destructor?**
- **Are class and structure the same? If not, what's the difference between a class and a structure?**
- **Explain Inheritance with an example?**
- **What is a subclass?**
- **Define a superclass?**
- **What is the difference between overloading and overriding?**
- **What is an interface?**
- **What is meant by static polymorphism?**
- **What is meant by dynamic polymorphism?**

Encapsulation in C++

In normal terms **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 manipulates them.
Consider a real life example of encapsulation, in a company there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keep records of all the data related to finance. Similarly the sales section handles all the sales related activities and keep records of all the sales. Now there may arise a situation when for some reason an official from finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of sales section and the employees that can manipulate them are wrapped under a single name "sales section".



Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.

In C++ encapsulation can be implemented using Class and access modifiers. Look at the below program:

```
// c++ program to explain
// Encapsulation

#include<iostream>
using namespace std;

class Encapsulation
{
    private:
        // data hidden from outside world
        int x;

    public:
        // function to set value of
        // variable x
        void set(int a)
        {
            x =a;

 // function to return value of
        // variable x
        int get()
        {
            return x;
        }
};

// main function
int main()
{
    Encapsulation obj;
```

```
obj.set(5);


cout<<obj.get();
return 0; }
```

output:

```
5
```

In the above program the variable **x** is made private. This variable can be accessed and manipulated only using the functions get () and set() which are present inside the class. Thus we can say that here, the variable x and the functions get () and set() are binded together which is nothing but encapsulation.

Role of access specifiers in encapsulation

As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

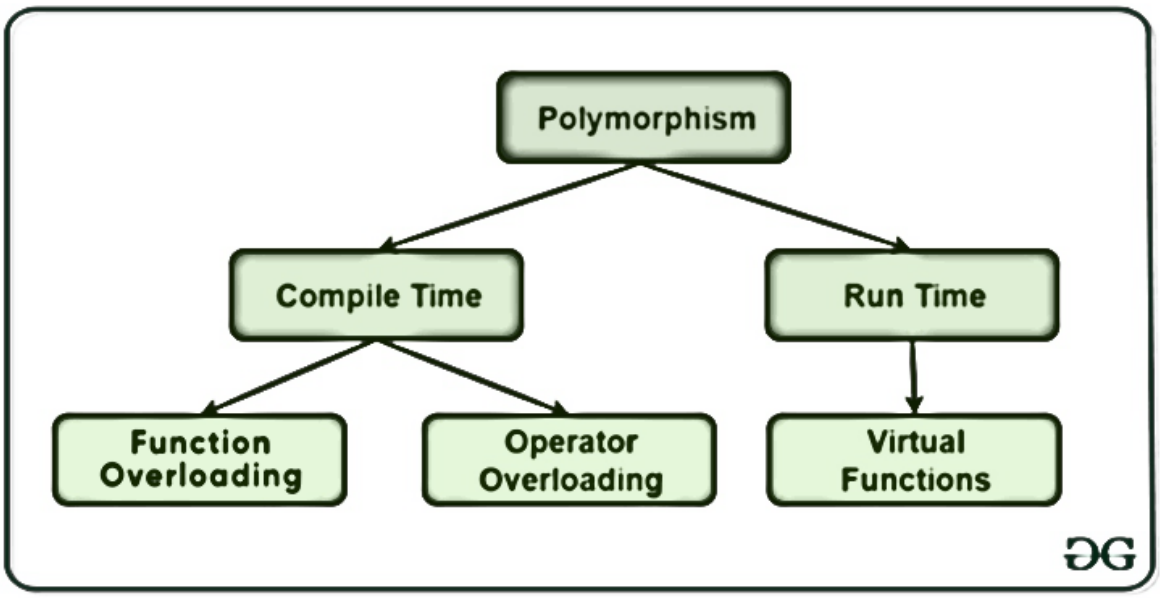
1. The data members should be labeled as private using the **private** access specifiers
2. The member function which manipulates the data members should be labeled as public using the **public** access specifier.

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. A real-life example of polymorphism, a person at the same time can have different characteristics. 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



Types of Polymorphism:

1. **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or 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**.

```
// C++ program for function overloading

#include <bits/stdc++.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 () {
    DEMO 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
```

In the above example, a single function named *func* acts differently in three different situations which is the property of polymorphism.

Operater 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.

```
// CPP program to illustrate

// 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.

Runtime polymorphism:

This type of polymorphism is achieved by Function Overriding.

- **Function overloading:** on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// C++ program for function overriding

#include <bits/stdc++.h>

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 () //print () is already virtual function in derived class, we could also declared as virtual void print () explicitly
    { cout<< "print derived class" <<endl; }
    void show ()
    { cout<< "show derived class" <<endl; }
}
//main function
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
    //virtual function, binded at runtime (Runtime polymorphism)
    bptr->print();
     // Non-virtual function, binded at compile time
    bptr->show();
    return 0;}
```

Output:

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

Abstraction in C++

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.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

Abstraction using Classes: We can implement Abstraction in C++ using classes. 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 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 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 power of numbers.

Abstraction using access specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class, can be accessed from anywhere in the program.

- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that defines the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

Example:

```
#include <iostream>

using namespace std;

class implementAbstraction

{

   private:

      int a, b;

   public:

      // method to set values of

      // private members

      void set(int x, int y)

      {

         a = x;

         b = y;

      }

      void display() {

cout<<"a = " <<a << endl;

cout<<"b = " << b << endl;

      }

};

int main()
```

```
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
   return 0;
}
```

Output:

```
a = 10
b = 20
```

You can see in the above program we are not allowed to access the variables a and b directly, however one can call the function set() to set the values in a and b and the function display() to display the values of a and b.

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.

Inheritance in C++

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 Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.

The article is divided into the following subtopics:

- Why and when to use inheritance?
- Modes of Inheritance
- Types of Inheritance

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown.

You can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:

Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

Implementing inheritance in C++: For creating a sub-class that is inherited from the base class we have to follow the below syntax.

Syntax:

```
class subclass_name :

access_mode base_class_name
{// body of subclass
};
```

Here, **subclass_name** is the name of the subclass, access mode as the mode in which you want to inherit the subclass for example public, private, etc. and **base_class_name** is the name of the base class from which you want to inherit the subclass.

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.

// C++ program to demonstrate implementation

// of Inheritance

```
#include <bits/stdc++.h>

using namespace std;
```

```
// Base class
class Parent {
public:
    int id_p;
};
// Sub class inheriting from Base Class(Parent)
class Child : public Parent {
public:
    int id_c;
};
// main function
int main()
{
    Child obj1;
// An object of class child has all data members
// and member functions of class parent
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is: " << obj1.id_c << '\n';
    cout << "Parent id is: " << obj1.id_p << '\n';

    return 0;
}
```

Output:

```
Child id is: 7

Parent id is: 91
```

In the above program, the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

Modes of Inheritance: There are 3 modes of inheritance.

1. **Public Mode**: If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
2. **Protected Mode**: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
3. **Private Mode**: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C, and D all contain the variables x, y, and z in the below example. It is just a question of access.

```
// C++ Implementation to show that a derived class

// doesn’t inherit access to private data members.

// However, it does inherit a full parent object.

class A {

public:

    int x;

protected:

    int y;

private:

    int z;

};

class B : public A {

    // x is public

    // y is protected

    // z is not accessible from B

};

class C : protected A {

    // x is protected

    // y is protected

    // z is not accessible from C

};

class D : private A // 'private' is default for classes

{

    // x is private

    // y is private

    // z is not accessible from D

};
```

The below table summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:

TYPES OF INHERITANCE IN C++:

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.

Syntax:

```
class subclass_name : access_mode base_class
{
// body of subclass
};
```

```
// C++ program to explain
// Single inheritance
#include<iostream>
using namespace std;
// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle\n";
    }
};
// sub class derived from a single base classes
class Car : public Vehicle {
};
// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle
```

2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.

Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2,
....
{
// body of subclass
};
```

Here, the number of base classes will be separated by a comma (', ') and the access mode for every base class must be specified.

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;
// first base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};
// second base class
class FourWheeler {
public:
   FourWheeler()
   {
      cout << "This is a 4 wheeler Vehicle\n";
   }
};
// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
};
// main function
int main()
{
   // Creating object of sub class will
```

```
 // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.

```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;
// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};
// first sub_class derived from class vehicle
class fourWheeler : public Vehicle {
public:
    fourWheeler()
    {
        cout << "Objects with 4 wheels are vehicles\n";
    }
};
// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler {
public:
    Car() { cout << "Car has 4 Wheels\n"; }
};
// main function
int main()
{
    // Creating object of sub class will
```

```
// invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;
// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};
// first sub class
class Car : public Vehicle {
};
// second sub class
class Bus : public Vehicle {
};
// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Car obj1;
    Bus obj2;
    return 0;
}
```

Output

```
This is a Vehicle
This is a Vehicle
```

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.
Below image shows the combination of hierarchical and multiple inheritances:

```
// C++ program for Hybrid Inheritance
#include <iostream>
using namespace std;
// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};
// base class
class Fare {
public:
   Fare() { cout << "Fare of Vehicle\n"; }
};
// first sub class
class Car : public Vehicle {
};
// second sub class
class Bus : public Vehicle, public Fare {
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base class.
   Bus obj2;
   return 0;
}
```

Output

```
This is a Vehicle
Fare of Vehicle
```

6. A special case of hybrid inheritance: Multipath inheritance:

A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.

```
// C++ program demonstrating ambiguity in Multipath
// Inheritance
#include <iostream>
using namespace std;
class ClassA {
public:
   int a;
};
class ClassB : public ClassA {
public:
   int b;
}
class ClassC : public ClassA {
public:
   int c;
};
class ClassD : public ClassB, public ClassC {
public:
   int d;
};
int main()
{
   ClassD obj;
   // obj.a = 10;              // Statement 1, Error
   // obj.a = 100;              // Statement 2, Error
   obj.ClassB::a = 10; // Statement 3
   obj.ClassC::a = 100; // Statement 4obj.b = 20;
   obj.c = 30;
```

```
 obj.d = 40;
   cout << " a from ClassB  : " << obj.ClassB::a;
   cout << "\n a from ClassC  : " << obj.ClassC::a;
   cout << "\n b : " << obj.b;
   cout << "\n c : " << obj.c;
   cout << "\n d : " << obj.d << '\n';
}
```

Output:

```
a from ClassB: 10
a from ClassC: 100
b : 20
c : 30
d : 40
```

In the above example, both ClassB and ClassC inherit ClassA, they both have a single copy of ClassA. However, Class-D inherits both ClassB and ClassC, therefore Class-D has two copies of Class A, one from ClassB and another from ClassC.

If we need to access the data member of ClassA through the object of Class-D, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bcoz compiler can't differentiate between two copies of ClassA in Class-D.

2 Way to avoid ambiguity:

1) Avoiding ambiguity using the scope resolution operator: Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

```
obj.ClassB::a = 10;     // Statement 3
obj.ClassC::a = 100;    // Statement 4
```

Note: Still, there are two copies of ClassA in Class-D.

2) Avoiding ambiguity using the virtual base class:

```
#include<iostream>
class ClassA
{
  public:
    int a;
};
class ClassB : virtual public ClassA
```

```
{
  public:
    int b;
};
class ClassC : virtual public ClassA
{
  public:
    int c;
};
class ClassD : public ClassB, public ClassC
{
  public:
    int d;
};
int main()
{
    ClassD obj;
    obj.a = 10;       // Statement 3
    obj.a = 100;      // Statement 4
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout << "\n a : " << obj.a;
    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

Output:

```
a : 100
b : 20
c : 30
d : 40
```

According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.

Inline Functions in C++

Inline function is one of the important features of C++. So, let's first understand why inline functions are used and what is the purpose of inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline function to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
    // function code
}
```

Remember, in lining is only a request to the compiler, not a command. Compiler can ignore the request for in lining. Compiler may not perform inlining in such circumstances like:
1)If a function contains a loop. (for, while, do-while)

2)If a function contains static variables.

3) If a function is recursive.

4) If a function return type is other than void, and the return statement doesn't exist in function body.

5) If a function contains switch or goto statement.

Inline functions provide following advantages:

1)Function call overhead doesn't occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:

1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

   2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

   3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

```
#include <iostream>

using namespace std;

inline int cube (int s)

{

   return s*s*s;

}

int main ()

{

   cout << "The cube of 3 is: " << cube (3) << "\n";

   return 0;

} //Output: The cube of 3 is: 27
```

Inline function and classes: It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.
For example:

```
class S

{

public:
```

```
inline int square(int s) // redundant use of inline
   {
      // this function is automatically inline
      // function body
   }
};
```

The above style is considered as a bad programming style. The best programming style is to just write the prototype of function inside the class and specify it as an inline in the function definition.
For example:

```
class S
{
public:
   int square(int s); // declare the function
};

inline int S::square(int s) // use inline prefix
{

}
```

The following program demonstrates this concept:

```
#include <iostream>
using namespace std;
class operation
{
   int a,b,add,sub,mul;
   float div;
public:
   void get();
   void sum();
   void difference();
   void product();
   void division();
};
```

```
inline void operation :: get()
{
   cout << "Enter first value:";
   cin >> a;
   cout << "Enter second value:";
   cin >> b;
}

inline void operation :: sum()
{
   add = a+b;
   cout << "Addition of two numbers: " << a+b << "\n";
}

inline void operation :: difference()
{
   sub = a-b;
   cout << "Difference of two numbers: " << a-b << "\n";
}

inline void operation :: product()
{
   mul = a*b;
   cout << "Product of two numbers: " << a*b << "\n";
}

inline void operation ::division()
{
   div=a/b;
   cout<<"Division of two numbers: "<<a/b<<"\n" ;
}

int main()
{
   cout << "Program using inline function\n";
```

```
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}
```

Output:

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

What is wrong with macro? Readers familiar with the C language knows that C language uses macro. The preprocessor replace all macro calls directly within the macro code. It is recommended to always use inline function instead of macro. According to Dr. Bjarne Stroustrup the creator of C++ that macros are almost never necessary in C++ and they are error prone. There are some problems with the use of macros in C++. Macro cannot access private members of class. Macros looks like function call but they are actually not.
Example:

```
#include <iostream>
using namespace std;
class S
{
    int m;
public:
#define MAC(S::m)    // error };
```

C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. Preprocessor macro is not capable for doing this. One other thing is that the macros are managed by preprocessor and inline functions are managed by C++ compiler.

Remember: It is true that all the functions defined inside the class are implicitly inline and C++ compiler will perform inline call of these functions, but C++ compiler cannot perform inlining if the function is virtual. The reason is call to a virtual function is resolved at runtime

instead of compile time. Virtual means wait until runtime and inline means during compilation, if the compiler doesn't know which function will be called, how it can perform inlining?

One other thing to remember is that it is only useful to make the function inline if the time spent during a function call is more compared to the function body execution time. An example where inline function has no effect at all:

```
inline void show()
{
    cout << "value of S = " << S << endl;
}
```

The above function relatively takes a long time to execute. In general function which performs input output (I/O) operation shouldn't be defined as inline because it spends a considerable amount of time. Technically inlining of show() function is of limited value because the amount of time the I/O statement will take far exceeds the overhead of a function call.

Depending upon the compiler you are using the compiler may show you warning if the function is not expanded inline. Programming languages like Java & C# doesn't support inline functions.
But in Java, the compiler can perform inlining when the small final method is called, because final methods can't be overridden by sub classes and call to a final method is resolved at compile time. In C# JIT compiler can also optimize code by inlining small function calls (like replacing body of a small function when it is called in a loop).

Last thing to keep in mind that inline functions are the valuable feature of C++. An appropriate use of inline function can provide performance enhancement but if inline functions are used arbitrarily then they can't provide better result. In other words don't expect better performance of program. Don't make every function inline. It is better to keep inline functions as small as possible.

Operator Overloading in C++

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

A simple and complete example

```
#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 << '\n'; }

};

int main()

{

   Complex c1(10, 5), c2(2, 4);

   Complex c3 = c1 + c2;

   c3.print();

}
```

Output:

```
12 + i9
```

What is the difference between operator functions and normal functions?

Operator functions are the same as normal functions. The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.

Following is an example of global operator function.

```
#include<iostream>

using namespace std;

class Complex {

private:
```

```
int real, imag;
public:
    Complex(int r = 0, int i = 0)  {real = r;   imag = i;}
    void print() { cout << real << " + i" << imag << '\n'; }

// The global operator function is made friend of this class so
// that it can access private members
friend Complex operator + (Complex const &, Complex const &);
};



Complex operator + (Complex const &c1, Complex const &c2)
{
     return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
    return 0;
}
```

Output

```
12 + i9
```

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
```

Why can't. (dot),::, ?: and size of be overloaded?

Important points about operator overloading

1) For operator overloading to work, at least one of the operands must be a user defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor).

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

```
#include <iostream>

using namespace std;

class Fraction

{

private:

   int num, den;

public:

   Fraction(int n, int d) { num = n; den = d; }


   // Conversion operator: return float value of fraction

   operator float() const {

      return float(num) / float(den);

   }

};


int main() {

   Fraction f(2, 5);

   float val = f;

   cout << val << '\n';

   return 0;

}
```

Output:

```
0.4
```

Overloaded conversion operators must be a member method. Other operators can either be member method or global method.

4) Any constructor that can be called with a single argument works as a conversion constructor, means it can also be used for implicit conversion to the class being constructed.

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

class Point
{
private:
   int x, y;
public:
   Point(int i = 0, int j = 0) {
      x = i;  y = j;
   }
   void print() {
      cout << "x = " << x << ", y = " << y << '\n';
   }
};

int main() {
   Point t(20, 20);
   t.print();
   t = 30;   // Member x of t becomes 30
   t.print();
   return 0;
}
```

Output:

```
x = 20, y = 20

x = 30, y = 0
```

Function Overloading in C++

Function overloading is a feature of object oriented programming where two or more functions can have the same name but different parameters.

When a function name is overloaded with different jobs it is called Function Overloading.

In Function Overloading "Function" name should be the same and the arguments should be different.

Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output:

```
Here is int 10

Here is float 10.1

Here is char* ten
```

How Function Overloading works?

- *Exact match*:- (Function name and Parameter)
- *If* a *not exact match is found*:-

  ->Char, Unsigned char, and short are promoted to an int.

  ->Float is promoted to double

- *If no match found*:

  ->C++ tries to find a match through the standard conversion.

- *ELSE ERROR :-(*
- Function overloading and return type
- Functions that cannot be overloaded in C++
- Function overloading and const keyword
- Function Overloading vs Function Overriding in C++

Shallow Copy and Deep Copy in C++

In general, creating a copy of an object means to create an exact replica of the object having the same literal value, data type and resources.

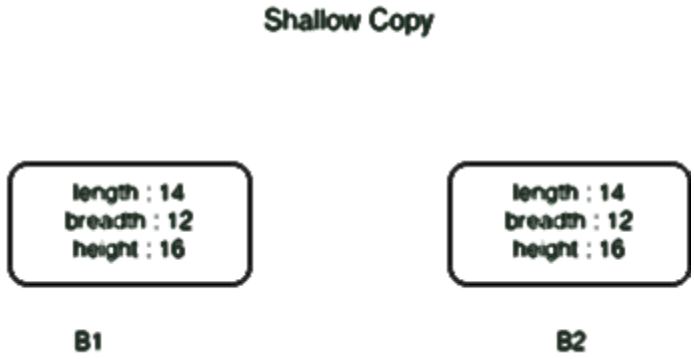
- Copy Constructor
- Default assignment operator

```
// Copy Constructor
demo Obj1(Obj);
or
demo Obj1 = Obj;

// Default assignment operator
demo Obj2;
Obj2 = Obj1;
```

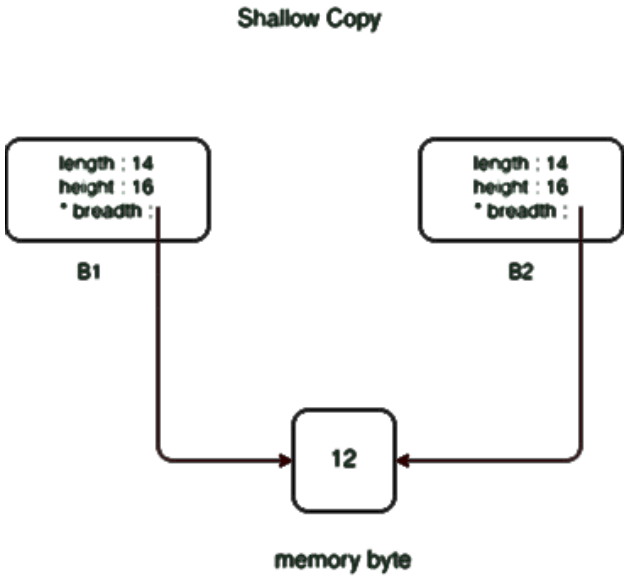
- Depending upon the resources like dynamic memory held by the object, either we need to perform **Shallow Copy** or **Deep Copy** in order to create a replica of the object. In general, if the variables of an object have been dynamically allocated then it is required to do a Deep Copy in order to create a copy of the object

Shallow object

In shallow copy, an object is created by simply copying the data of all variables of the original object. This works well if none of the variables of the object are defined in the heap section of memory. If some variables are dynamically allocated memory from heap section, then copied object variable will also reference then same memory location.
This will create ambiguity and run-time errors dangling pointer. Since both objects will reference to the same memory location, then change made by one will reflect those change in another object as well. Since we wanted to create a replica of the object, this purpose will not be filled by Shallow copy.
Note: C++ compiler implicitly creates a copy constructor and overloads assignment operator in order to perform shallow copy at compile time.



Shallow Copy of object if some variables are defined in heap memory, then:



Below is the implementation of the above approach:

```
// C++ program for the above approach
#include <iostream>
using namespace std;
// Box Class
class box {
private:
    int length;
    int breadth;
    int height;
public:
    // Function that sets the dimensions
    void set_dimensions(int length1, int breadth1,
                        int height1)
    {
        length = length1;
        breadth = breadth1;
        height = height1;
    }
    // Function to display the dimensions
    // of the Box object
    void show_data()
    {
        cout << " Length = " << length
```

```
            << "\n Breadth = " << breadth
            << "\n Height = " << height
            << endl;
    }};
// Driver Code
int main()
{
    // Object of class Box
    box B1, B3;
    // Set dimensions of Box B1
    B1.set_dimensions(14, 12, 16);
    B1.show_data();
    // When copying the data of object
    // at the time of initialization
    // then copy is made through
    // COPY CONSTRUCTOR
    box B2 = B1;
    B2.show_data();
    // When copying the data of object
    // after initialization then the
    // copy is done through DEFAULT
    // ASSIGNMENT OPERATOR
    B3 = B1;
    B3.show_data();
```

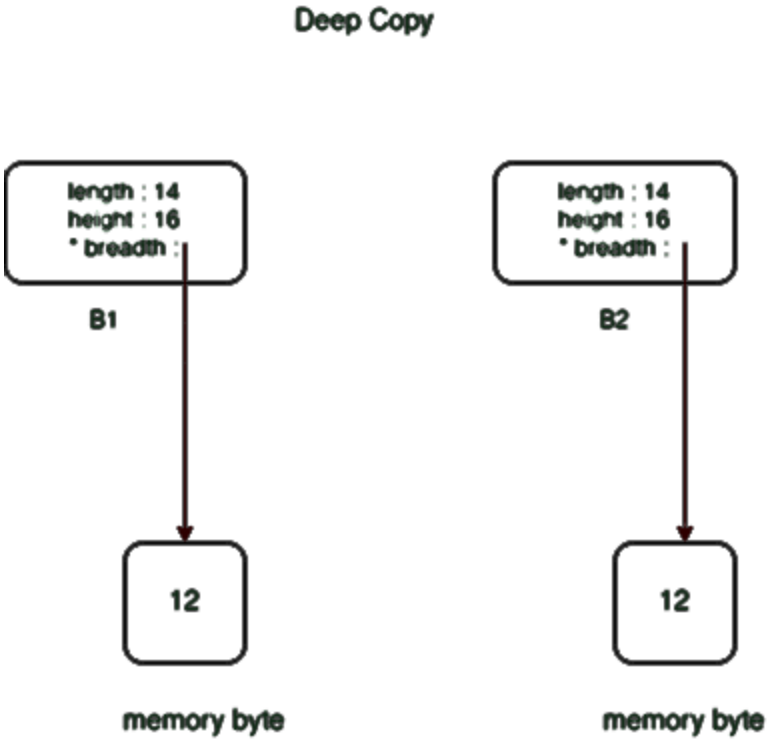
```
 return 0;

}
```

Output:

```
Length = 14
 Breadth = 12
 Height = 16
 Length = 14
 Breadth = 12
 Height = 16
 Length = 14
 Breadth = 12
 Height = 16
```

Deep Copy:

In Deep copy, an object is created by copying data of all variables and it also allocates similar memory resources with the same value to the object. In order to perform Deep copy, we need to explicitly define the copy constructor and assign dynamic memory as well if required. Also, it is required to dynamically allocate memory to the variables in the other constructors, as well.



Below is the implementation of the above approach:

```
// C++ program to implement the
// deep copy
#include <iostream>
using namespace std;
// Box Class
class box {
private:
    int length;
    int* breadth;
    int height;
public:
    // Constructor
```

```
box()
{
    breadth = new int;
}
// Function to set the dimensions
// of the Box
void set_dimension(int len, int brea,
                   int heig)
{
    length = len;
    *breadth = brea;
    height = heig;
}
// Function to show the dimensions
// of the Box
void show_data()
{
    cout << " Length = " << length
         << "\n Breadth = " << *breadth
         << "\n Height = " << height
         << endl;
}
// Parameterized Constructors for
// for implementing deep copy
```

```
    box(box& sample)
    {
        length = sample.length;
        breadth = new int;
        *breadth = *(sample.breadth);
        height = sample.height;
    }
    // Destructors
    ~box()
    {
        delete breadth;
    }
};
// Driver Code
int main()
{
    // Object of class first
    box first;
    // Set the dimensions
    first.set_dimension(12, 14, 16);
    // Display the dimensions
    first.show_data();
    // When the data will be copied then
    // all the resources will also get
```

```
    // allocated to the new object

    box second = first;

    // Display the dimensions

    second.show_data();

    return 0;

}
```

Output:

```
Length = 12
 Breadth = 14
 Height = 16
 Length = 12
 Breadth = 14
 Height = 16
```

Application of OOPs in C++

OOPs stands for Object-Oriented Programming. It is about creating objects that contain both data and functions. Object-Oriented programming has several advantages over procedural languages. As OOP is faster and easier to execute it becomes more powerful than procedural languages like C++. OOPs is the most important and flexible paradigm of modern programming. It is specifically useful in modeling real-world problems. Below are some applications of OOPs:

- **Real-Time System design:** Real-time system inherits complexities and makes it difficult to build them. OOP techniques make it easier to handle those complexities.
- **Hypertext and Hypermedia:** Hypertext is similar to regular text as it can be stored, searched, and edited easily. Hypermedia on the other hand is a superset of hypertext. OOP also helps in laying the framework for hypertext and hypermedia.
- **AI Expert System:** These are computer application that is developed to solve complex problems which are far beyond the human brain. OOP helps to develop such an AI expert System
- **Office automation System:** These include formal as well as informal electronic systems that primarily concerned with information sharing and communication to and from people inside and outside the organization. OOP also help in making office automation principle.

- **Neural networking and parallel programming:** It addresses the problem of prediction and approximation of complex-time varying systems. OOP simplifies the entire process by simplifying the approximation and prediction ability of the network.
- **Stimulation and modeling system:** It is difficult to model complex systems due to varying specifications of variables. Stimulating complex systems require modeling and understanding interaction explicitly. OOP provides an appropriate approach for simplifying these complex models.
- **Object-oriented database:** The databases try to maintain a direct correspondence between the real world and database object in order to let the object retain it identity and integrity.
- **Client-server system:** Object-oriented client-server system provides the IT infrastructure creating object-oriented server internet(**OCSI**) applications.
- **CIM/CAD/CAM systems:** OOP can also be used in manufacturing and designing applications as it allows people to reduce the efforts involved. For instance, it can be used while designing blueprints and flowcharts. So it makes it possible to produce these flowcharts and blueprint accurately.

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

CREDITS- INTERNET.

DISCLOSURE- ALL THE DATA AND IMAGES ARE TAKEN FROM GOOGLE AND INTERNET.

REFRENCES USED-

1) Effective C++, Scott Meyers

2) http://www.parashift.com/c++-faq/inline-and-perf.html

3) http://www.cplusplus.com/forum/articles/20600/

4) Thinking in C++, Volume 1, Bruce Eckel.

5) C++ the complete reference, Herbert Schildt

6) http://en.wikipedia.org/wiki/Operator_overloading

FOLLOW ME FOR PART-2 OF THIS NOTES AND ALSO FOR MORE AMAZING INFORMATIVE NOTES AND CONTENS.

GITHIUB REPO FOR ALL NOTES - GitHub - himanshukumar9/Placement_preparation_exclusive_notes

THANKYOU