# OOPs Basic

There are several kinds of major programming paradigms:

- Logical
- Functional/Procedural
- Object-Oriented

What is OOPs?

Oops-object oriented programming is a programming paradigm which is based on objects, instead of just functions. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism in programming. The main aim of OOP is to bind together the data and the functions .it's prevent the accessibility of the data from outside function.

OOPs advantage:

1. Code can be reused through inheritance
2. Data and function are bound together by encapsulation.so data can protected from non member function
3. Real world entities like abstraction, encapsulation, inheritance, and polymorphism are implemented by using OOPs.
4. Class hierarchies are helpful in the design process allowing increased extensibility.
5. Modularity is achieved. Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem

Disadvantage:

- To Program with OOP, Programmer need proper skills such as design skills programming skills, thinking in term of objects etc.

S in OOPs:

According to Wikipedia ,There is no S in OOPs.

But some other forums in google claim that S means System or Structure.

Procedural vs oop:

| Procedural Oriented Programming | Object Oriented Programming |
|---|---|
| In procedural programming, program is divided into small parts called functions. | In object oriented programming, program is divided into small parts called objects. |
| There is no access specifier in procedural programming. | Object oriented programming have access specifiers like private, public, protected etc. |
| Adding new data and function is not easy. | Adding new data and function is easy. |
| Procedural programming does not have Data Hiding concept.so its not secure | Object oriented programming provides data hiding .so it is more secure. |
| In procedural programming, overloading is not possible. | Overloading is possible in object oriented programming. |
| Examples: C, FORTRAN, Pascal, Basic etc. | Examples: C++, Java, Python, C# etc. |

Object Based languages vs Object Oriented Languages:

- Object Based Languages does not support inheritance, polymorphism.
- Object based languages does not supports built-in objects.
- EX. JavaScript, VB

**Class**:

A Class is a user defined data-type which has data members and member functions. Data members and member functions defines the state and behaviour of the objects in a Class. class is like a blueprint for an object. Class does not take any space.

Ex consider the student class

the data member will be id,name ,subject marks and member functions can be total marks.

**Object**:

An Object is an instance (copy/creating reference) of a Class. When a class is defined, no memory is allocated. Memory is allocated during the object creation.

Ex. Student1, student2 are the object of above example

## OOP Principles/ Features

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

One real time example which has all the features of oops
Lets consider olden days telephone(Rotary dial telephone) and latest mobile phone(Samsung s9)
**Abstraction**-show important information to user and hide unwanted details
Latest phone ,contact application shows the contact details to user and hide the detail about how the contact stored and how retrieved
**Encapsulation-** bind everything together into single unit.
Hardware and software bind together into a single unit called phone
**Inheritance**-to accure the feature from others
Lets consider the whats up app.when u install the app it will ask the permission For camera and contacts etc.here,the feature of camera and contacts accure from other app.
**Polymorphism**-poly means many form.
A mobile phone can act as a camera,video recorder, writing pad,and call to others

## Abstraction

Abstraction defined as displaying only the important information and hiding the implementation details. . Abstraction is also called as data hiding.

Ex.1 when you make your email id or login into an email id, you can only type the login Id and password and view it. You do not know how the data is being verified and how it is being stored

Ex.2 when we make phone call to someone is an example of Abstraction .we only press the button and call connected to other person.we exactly don't know the inner mechanism of this process.

Data abstraction can be achieved through Abstract class.

Abstract class:

An abstract class is a class that consists of abstract methods. These methods are basically declared but not defined. If these methods are to be used in some subclass, they need to be exclusively defined in the subclass.

In c++,

```
class Base
{
    public:
    virtual void show() = 0;    // Pure Virtual Function
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Implementation of Virtual Function in Deri
    }
};
```

In the above example we can see WHAT function an implementing class will contain but we cannot see HOW the function is implemented.

Interfaces in C++: Abstract Class

In C++, we use terms abstract class and interface interchangeably. A class with pure virtual function is known as abstract class. For example the following function is a pure virtual function:

virtual void fun() = 0;

You can call this function an abstract function as it has no body. The derived class must give the implementation to all the pure virtual functions of parent class else it will become abstract class by default. In C++, an interface can be simulated by making all methods as pure virtual. In Java, there is a separate keyword for interface.

## Why we need a abstract class?

Let's understand this with the help of a real life example. Lets say we have a class Animal, animal sleeps, animal make sound, etc. For now I am considering only these two behaviours and creating a class Animal with two functions

sound() and sleeping(). Now, we know that animal sounds are different cat says “meow”, dog says “woof”. So what implementation do I give in Animal class for the function sound(), the only and correct way of doing this would be making this function pure abstract so that I need not give implementation in Animal class but all the classes that inherits Animal class must give implementation to this function. This way I am ensuring that all the Animals have sound but they have their unique sound.

## Abstract class Example

```
#include<iostream>
using namespace std;
class Animal{
public:
   //Pure Virtual Function
   virtual void sound() = 0;

   //Normal member Function
   void sleeping() {
      cout<<"Sleeping";
   }
};
class Dog: public Animal{
public:
   void sound() {
      cout<<"Woof"<<endl;
   }
};
int main(){
   Dog obj;
   obj.sound();
   obj.sleeping();
   return 0;
}
```

Rules of Abstract Class

1) Any class that has a pure virtual function is an abstract class.

2) We cannot create the instance of abstract class. For example: If I have written this line Animal obj; in the above program, it would have caused compilation error.

3) We can create pointer and reference of base abstract class points to the instance of child class. For example, this is valid:

Animal *obj = new Dog();obj->sound();

4) Abstract class can have constructors.

5) If the derived class does not implement the pure virtual function of parent class then the derived class becomes abstract.

## Encapsulation

The process of binding data and corresponding methods (behaviour) together into a single unit is called encapsulation. This is to prevent the access to the data directly from outside of the class. Encapsulation enables the Security through data hiding and Abstraction.

How Encapsulation is achieved in a class

To do this:

1) Make all the data members private.

2) Create public setter and getter functions for each data member in such a way that the set function set the value of data member and get function get the value of data member.

Real-time Example:

Suppose you have an account in the bank. If your balance variable is declared as a public variable in the bank software, your account balance will be known as public, In this case, anyone can know your account balance. So, would you like it? Obviously No.
So, they declare balance variable as private for making your account safe, so that anyone cannot see your account balance. The person who has to see his account balance, he will have to access private members only through methods defined inside that class and this method will ask your account holder name or user Id, and password for authentication.
Thus, We can achieve security by utilizing the concept of data hiding. This is called Encapsulation.

Ex.2

Let company have two different section as finance section, sales section .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. Here sales data and employee of sales department binned together as a single name called sales section.

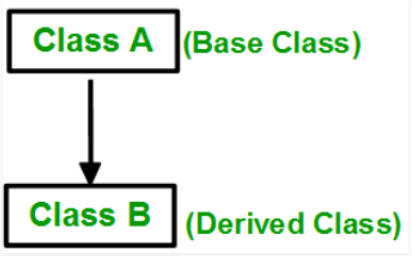
## Inheritance

Inheritance allows the child class to acquire the properties (the data members) and behaviour (the member functions) of parent class. The main advantages of inheritance are code reusability and readability.

Here we have two classes Teacher and MathTeacher, the MathTeacher class inherits the Teacher class which means Teacher is a parent class and Math Teacher is a child class. The child class can use the property collegeName of parent class.

Real time example: in real world, children have some characters which are similar to their parents or grandparents.

Types of Inheritance in C++

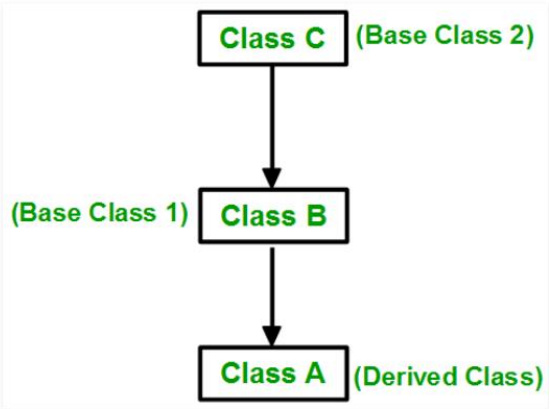



1) Single inheritance

In Single inheritance one class inherits one class exactly.

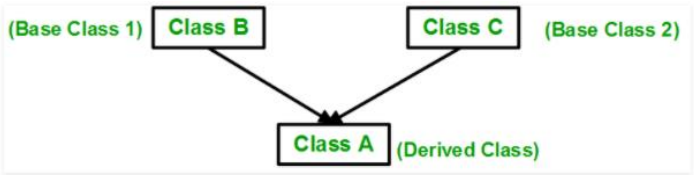
2) Multilevel inheritance

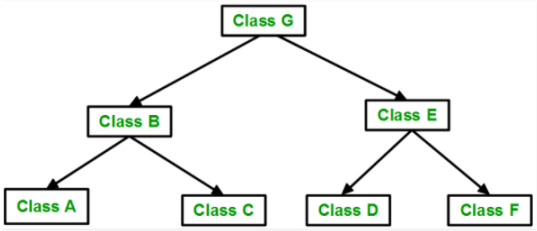
In this type of inheritance one class inherits another child class.




3) Multiple inheritance

A class can inherit more than one class. This means that in this type of inheritance a single child class can have multiple parent classes.



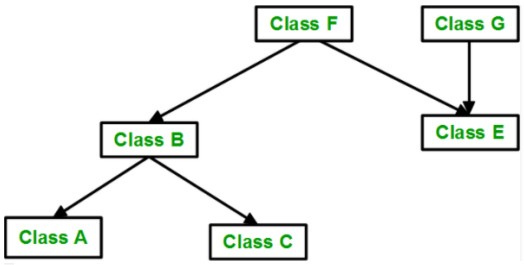




4) Hierarchical inheritance

In this type of inheritance, one parent class has more than one child class.

5) Hybrid inheritance

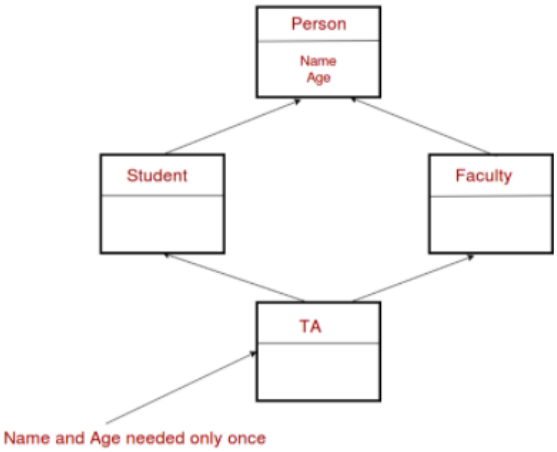
Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.




## 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. There are 2 ways to avoid this ambiguity:

- **Use scope resolution operator**
- **Use virtual base class**




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

```
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;
};
```

## Polymorphism

Polymorphism refers to the ability of a function to exist in multiple forms.

Take example of your mother, she is one person but she has different roles to perform like being someone's sister, wife, mother, aunt, employee. So the same person possess different behaviour in different situations. This is called polymorphism.

Types of Polymorphism:

- Compile time polymorphism
- Run time polymorphism

Compile time polymorphism (static):

Compile time 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.

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

int absolute(int);
float absolute(float);

int main() {
     int a = -5;
     float b = 5.5;

     cout << "Absolute value of " << a << " = " << absolute(a) << endl;
     cout << "Absolute value of " << b << " = " << absolute(b);
     return 0;
}

int absolute(int var) {
       if (var < 0)
            var = -var;
     return var;
}

float absolute(float var){
     if (var < 0.0)
          var = -var;
     return var;
}
```

Operator Overloading

C++ also provides option to overload operators. We know that the additional operator('+') whose task is to add two operands. Therefore, a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

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

class Test
{
    private:
        int count;

    public:
         Test(): count(5){}

         void operator ++()
         {
             count = count+1;
         }
         void Display() { cout<<"Count: "<<count; }
};

int main()
{
     Test t;
     // this calls "function void operator ++()" function
     ++t;
     t.Display();
     return 0;
}
```

Run time polymorphism (dynamic):

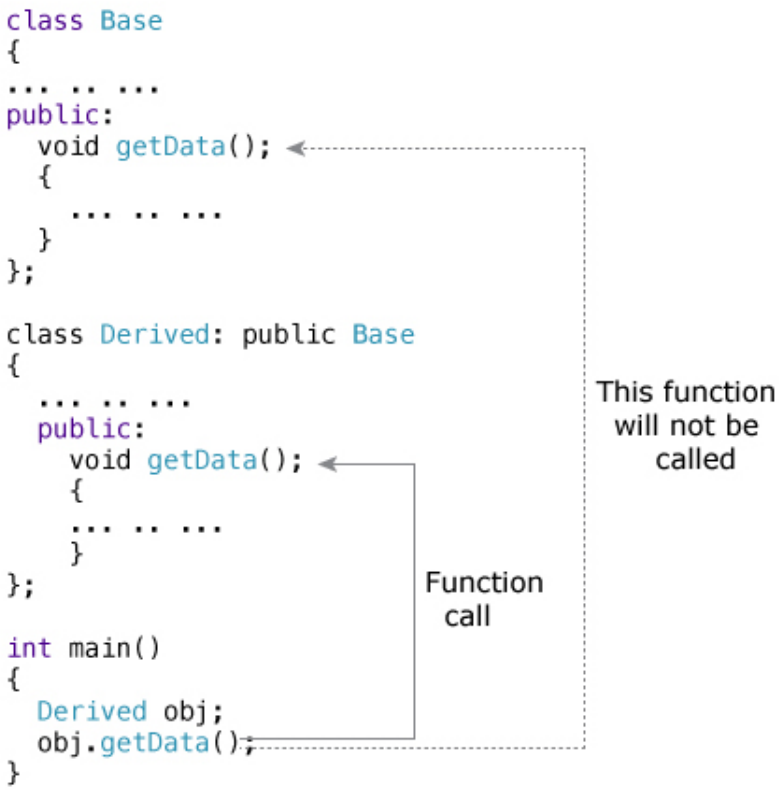
Runtime polymorphism is achieved by function overriding. Function overriding occurs when a derived class has a definition for one of the member functions of the base class. The base function is said to be overridden.

**List of operators that can be overloaded are:**

```
+    -    *    /      %        ^
&    |    ~    !,        =
    =       ++         --
    ==     !=       &&         ||
+=     -=     /=     %=       ^=         &=
|=     *=     =       []         ()
->     ->*     new     new []      delete     delete []
```

List of operators that cannot be overloaded

```
1> Scope Resolution Operator  (::)
2> Pointer-to-member Operator (.*)
3> Member Access or Dot operator  (.)
4> Ternary or Conditional Operator (?:)
5> Object size Operator   (sizeof)
6> Object type Operator   (typeid)
```




Run time polymorphism also achieved by virtual functions and pointers.

**Static binding vs dynamic binding:**

Binding generally refers to a mapping of one thing to another. In the context of compiled languages, binding is the link between a function call and the function definition.

There are two types of binding in C++: static (early) binding and dynamic (late) binding.

The static binding happens at the compile-time and dynamic binding happens at the runtime. Hence, they are also called early and late binding respectively.

In static binding, the function definition and the function call are linked during the compile-time whereas in dynamic binding the function calls are not resolved until runtime.

Static binding can be achieved using the normal function calls, function overloading and operator overloading while dynamic binding can be achieved using the virtual functions.

Static binding:

```
class ComputeSum
{
    public:

    int sum(int x, int y)
    {
        return x + y;
    }

    int sum(int x, int y, int z)
    {
        return x + y + z;
    }
};

int main()
```

Dynamic Binding:

```
// C++ program to illustrate the concept of dynam
#include <iostream>
using namespace std;

class B
{
    public:

    // Virtual function
    virtual void f()
    {
        cout << "Base class function called.\n";
    }
};

class D: public B
{
    public:
    void f()
    {
        cout << "Derived class function called.\n
    }
};

int main()
{
    B base;
    D derived;

    B *basePtr = &base;
    basePtr->f();

    basePtr = &derived;
    basePtr->f();

    return 0;
}
```

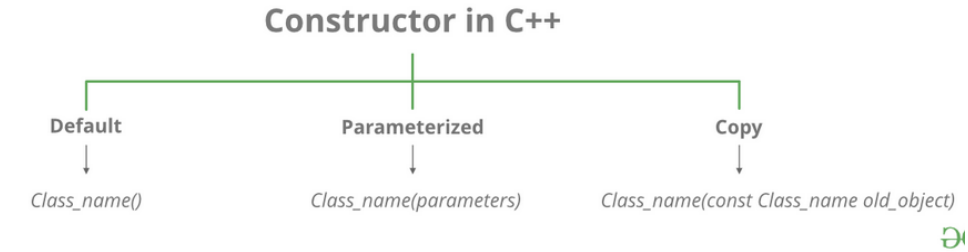
```
Output:

Base class function called.
Derived class function called.
```

## Message passing:

Objects communicate with one another by sending and receiving information to each other. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

## 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. Constructors have the same name as the class and may be defined inside or outside the class definition. •Constructor must have no return type.

### Default constructors

1. **Default Constructors:** Default constructor is the constructor which

```cpp
// Cpp program to illustrate the
// concept of 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
```

## Parametrized constructors

```cpp
// CPP program to illustrate
// parameterized constructors
#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;
}
```

## Copy constructors

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

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

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

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

## Uses of Parameterized constructor:

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

## Can we have more than one constructors in a class?

Yes, It is called Constructor Overloading.

```cpp
// C++ program to illustrate
// Constructor overloading
#include <iostream>
using namespace std;

class construct
{

public:
    float area;

    // Constructor with no parameters
    construct()
    {
        area = 0;
    }

    // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};

int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

Output:

```
0
200
```

## Destructors

Destructor is a member function which destructs or deletes an object.

```
class String
{
private:
    char *s;
    int size;
public:
    String(char *); // constructor
    ~String();      // destructor
};

String::String(char *c)
{
    size = strlen(c);
    s = new char[size+1];
    strcpy(s,c);
}

String::~String()
{
    delete []s;
}
```

### How destructors are different from a normal member function?

1. Destructors have same name as the class preceded by a tilde (~)
2. Destructors don't take any argument and don't return anything

### Can there be more than one destructor in a class?

- No, there can only one destructor in a class with class name preceded by ~, no parameters and no return type.

### When do we need to write a user-defined destructor?

If we do not write our own destructor in class, compiler creates a default destructor for us. When a class contains a pointer or dynamically allocated memory, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

### Can a destructor be virtual?

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function.

Virtual Destructor

Deleting a derived class object using a base class pointer that has a non-virtual destructor results in undefined behaviour. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behaviour.

```
// causing undefined behavior
#include<iostream>

using namespace std;

class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Although the output of following program may be different on different compilers,

```
Constructing base
Constructing derived
Destructing base
```

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example,

```
// A program with virtual destructor
#include<iostream>

using namespace std;

class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Output:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

## Inline function:

Inline function is optimization technique used by the compilers especially to reduce the execution time. Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime. if function is big (in term of executable instruction etc) then, compiler can ignore the “inline” request and treat the function as normal function.

WHEN THE FUNCTION IS CALLED COMPILER NEEDS SOME EXTRA TIME TO EXECUTE IT.

BUT IF OUR FUNCTION IS SMALL
AND WE ARE JUST DOING COMMON ADDITION
OR MULTIPLICATIONS IN FUNCTION

THEN WE CAN MAKE THAT FUNCTION INLINE

WE CAN MAKE FUNCTIONS INLINE BY JUST ADDING INLINE KEYWORD IN FRONT OF FUNCTION

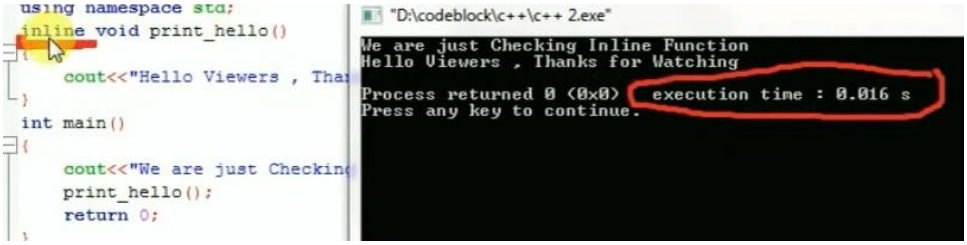
```
void print_hello()
{
cout<<"Hello World !"<<endl;
}
```

make it inline

```
inline void print_hello()
{
cout<<"Hello World !"<<endl;
}
```

```
#include <iostream>
using namespace std;
void print_hello()
{
    cout<<"Hello Viewers , Thanks f
}
int main()
{
    cout<<"We are just Checking Inl
    print_hello();
    return 0;
}
```

```
We are just Checking Inline Function
Hello Viewers , Thanks for Watching
Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```




INLINE IS JUST A KEYWORD ITS NOT AN COMMAND

INLINE IS A REQUEST

COMPILER DECIDES THE FUNCTION TO BE INLINE OR NOT

SITUATIONS WHERE INLINE EXPANSION MAY NOT WORK

IF A FUNCTION RETURNING VALUES, IF A LOOP , A SWITCH OR GOTO STATEMENT EXISTS.

IF FUNCTION CONTAINS STATIC VARIABLES

IF FUNCTION IS RECURSIVE

```cpp
#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 vs macro:

The inline functions are expanded during compilation, and the macros are expanded by the pre-processor.

Before compilation, the program is examined by the pre-processor and where ever it finds the macro in the program, it replaces that macro by its definition. Hence, the macro is considered as the “text replacement”.

```
#include <stdio.h>
#define GREATER(a, b) ((a < b) ? b : a)
int main( void)
{
  cout << "Greater of 10 and 20 is " << GREATER("20", "10") << "\n";
    return 0;
}
```

```
#include<stdio.h>
#define SQUARE(X)  X*X
int square(int x)
 {
   return (x*x);
 }
void main()

 {

 int x=3;
 int y=2;


 printf("Macro: %d\n",SQUARE(x+y));
 printf("Function: %d\n",square(x+y));

 }
```

There is no type checking in macro. But there is type checking in function.

## Friend function:

One of the important concepts of OOP is data hiding, i.e., a non-member 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 a friend class.

Friend function can access the private and protected data of a class.

It's non-member function of a class. Class cannot control the scope of the friend function. Friend function can be declared under private or public.

Inheritance: Reusability & extension ability

Friend class: reusability

```
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.
    ... .. ...
}
```

```
#include <iostream>
using namespace std;
class Box
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

**Output:**

```
Length of box: 10
```

the function is friendly to two classes:

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

class B;
class A {
    private:
      int numA;
    public:
      A(): numA(12) { }
      friend int add(A, B);
};

class B {
    private:
       int numB;
    public:
       B(): numB(1) { }
       friend int add(A , B);
};

// Function add() is the friend function of classes A and B
int add(A objectA, B objectB)
{
   return (objectA.numA + objectB.numB);
}

int main()
{
    A objectA;
    B objectB;
    cout<<"Sum: "<< add(objectA, objectB);
    return 0;
}
```

Similarly, like a friend function, a class can also be made a friend of another class

```
... .. ...
class B;
class A
{
   // class B is a friend class of class A
   friend class B;
   ... .. ...
}

class B
{
   ... .. ...
}
```

When a class(B) is made a friend class(A), all the member functions of that class(B) becomes friend functions of class(A).

In this program, all member functions of class B will be friend functions of class A. Thus, any member function of class B can access the private and protected data of class A. But, member functions of class A cannot access the data of class B.

Note:

2) Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.

3) Friendship is not inherited .

4) The concept of friends is not there in Java.

Inheritance and friendship

In C++, friendship is not inherited. If a base class has a friend function, then the function doesn't become a friend of the derived class(es).

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

class A
{
  protected:
    int x;
  public:
    A() { x = 0;}
  friend void show();
};

class B: public A
{
  public:
    B() : y (0) {}
  private:
    int y;

};

void show()
{
  B b;
  cout << "The default value of A::x = " << b.x;

  // Can't access private member declared in class 'B'
  cout << "The default value of B::y = " << b.y;
}

int main()
{
  show();
  getchar();
  return 0;
}
```

## Virtual function:

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. They are mainly used to achieve Runtime polymorphism

Functions are declared with a virtual keyword in base class.

```
#include <iostream>
using namespace std;
class A
{
   int x=5;
    public:
    void display()
    {
        std::cout << "Value of x is : " << x<<std::endl;
    }
};
class B: public A
{
    int y = 10;
    public:
    void display()
    {
        std::cout << "Value of y is : " <<y<< std::endl;
    }
};
int main()
{
    A *a;
    B b;
    a = &b;
   a->display();
    return 0;
}
output:
Value of x is : 5
```

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;   //pointer of base class
 B b;    //object of derived class
 a = &b;
 a->display();  //Late Binding occurs
}
```

**Output:**

```
Derived Class is invoked
```

## Exception:

An exception is a problem that arises during the execution of a program. It is occur during run time

## Class vs structure:

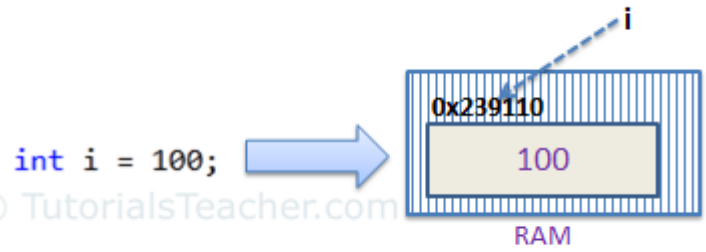
Class and structure both are user defined data type.class can contain the data member and member function.but structure can only contain data member.Members of a class are private by default and members of struct are public by default.

| Class | Structure |
|---|---|
| It is a reference data type and uses the keyword "class". | It is a value data type and uses the keyword "struct". |
| Object for a class is created in the heap memory. | Object for a structure is created in the stack memory. |
| We can always inherit another class. i.e., the concept of inheritance is applied . | Structures can never be inherited. |

Value vs reference type:

data types are categorized based on how they store their value in the memory.

Value type:




Memory allocation for Value Type

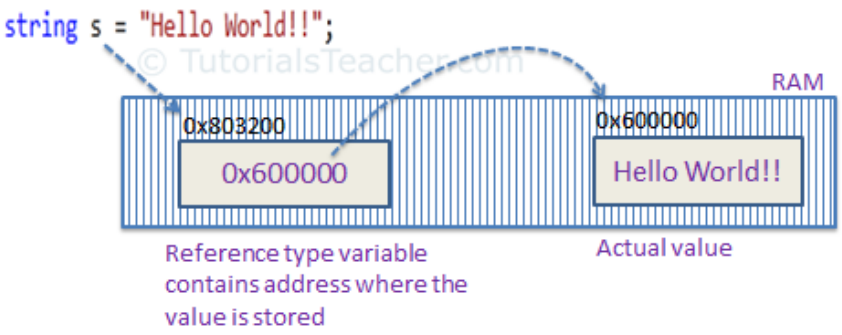
For example, consider integer variable int i = 100;

The system stores 100 in the memory space allocated for the variable 'i'.

Reference Type

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

string s = "Hello World!!";




Memory allocation for Reference type

Value type stores the value in its memory space, whereas reference type stores the address of the value where it is stored.

Primitive data types and struct are of the 'Value' type. Class objects, string, array, are reference types.

Value type passes by val by default. Reference type passes by ref by default.

Value types and reference types stored in Stack and Heap in the memory depends on the scope of the variable.

## Association, Dependency, Generalization

### Association

Association is a relationship between two objects. The relationship may be one-to-one, one-to-many, many-to-one, many-to-many .Aggregation is a special form of association. Composition is a special form of aggregation.

Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

Composition implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

### Dependency

A relationship between two objects where changing one may affect the other.

### Generalization

inheritance is a kind of generalization. We define it simply with a base class

having some properties, functions and so on. A new class will be derived from this base class and the child class will have access to all the functionality of the base or parent class.
It is also referred to as a “is-a" relationship; For example: we have a base class named Vehicle and a derived class named Car, from the base class. So Car will have access to all the functions, properties and so on of the base class Vehicle.(car is a vehicle)

All the best...

With