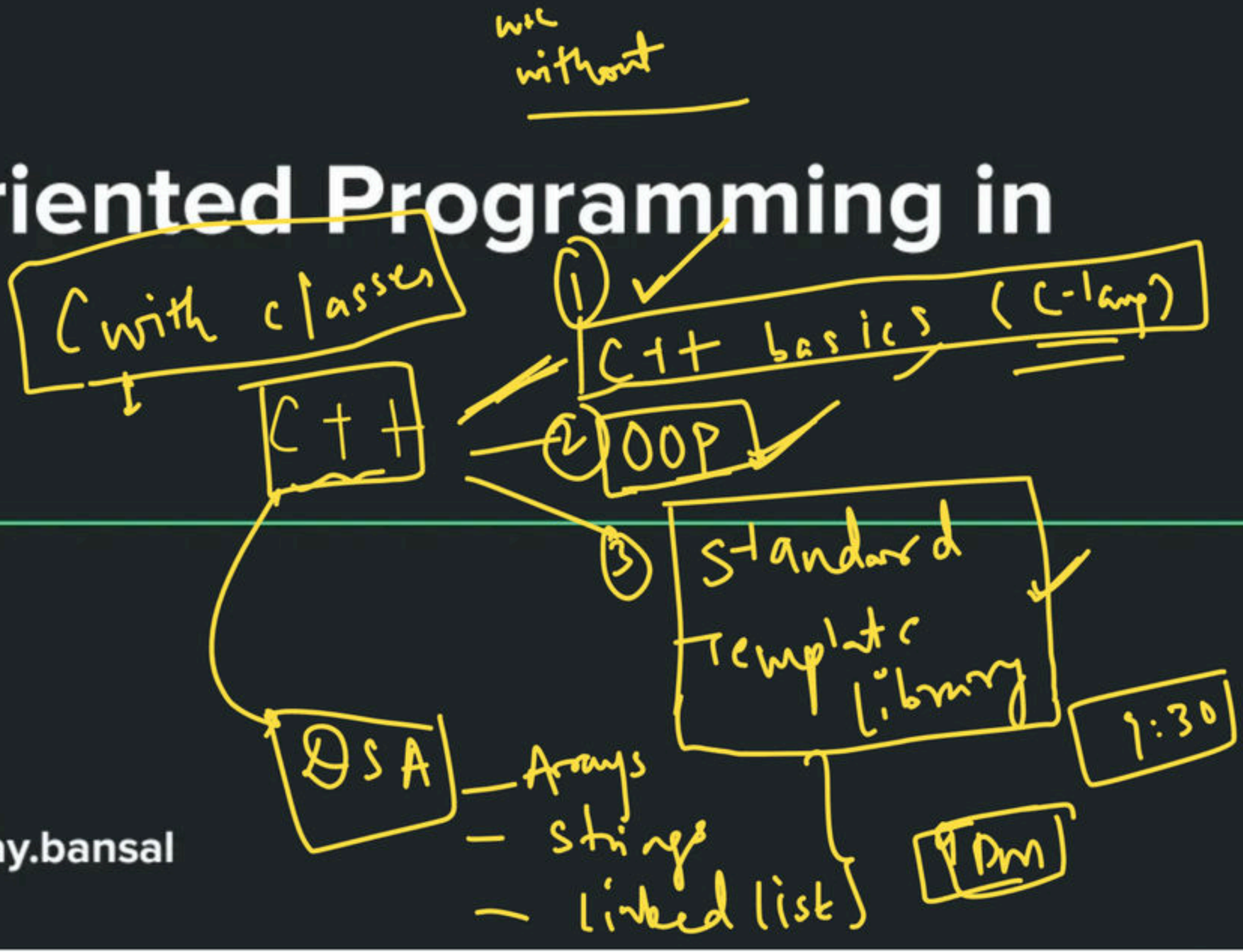
Quick Revision & Download C++ OOPs Slides (Lec 15-33)

Special class

Jay Bansal • June 13, 2021

Object Oriented Programming in C++



Jay Bansal

unacademy.com/@jay.bansal

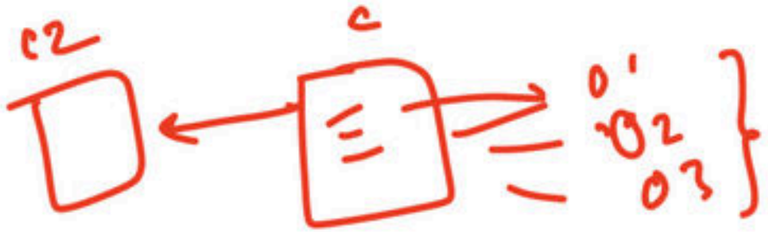
Object Oriented programming in C++

Object Oriented programming in C++

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

What is OOP ?



- Object Oriented Programming is the principle of design & development of programs using modular approach.
- The procedural programming focuses on processing of instructions in order to perform a desired computation, it emphasizes more on doing things like algorithms. Used in programming languages like C & Pascal.
- OOP combines both the data and the functions that operate on that data into a single unit called the object. It follows bottom-up design technique, it is piecing together the smaller systems to give rise to more complex systems.
- Another major component that plays a major role in OOP is the class. It is a template that represents a group of objects which share common properties and relationships.

C++ as Object Oriented Language ?

- It is an object-oriented, general-purpose programming language, derived from C.
- Existing code on C can be used with C++, hence mode compatible. Even existing pre-compiled libraries can be used with new C++ code.
- Also there is no additional cost for using C++, hence efficient.
- Major improvements over C:
  - Stream I/O
  - Strong typing
  - Inlining
  - Default argument values
  - Parameter passing by reference
  - OOP advantages.

OOP Principles ?

- Object Oriented Programming is a methodology characterized by the following concepts:

- **Encapsulation**: The process of binding data members (variables, properties) and member functions (methods) into a single unit. A class is the best example.
  - Take an example of a pharmacy store.
  - You go to the shop and ask for a medicine.
  - There only the chemist has the access to the medicines of the store and he knows what medicine to give you.
  - This reduces the risk of taking a medicine that is not prescribed to you.
  - Here medicines are the member variables, chemist is the member function and you are the external piece of code.

OOP Principles ?

- **Abstraction**: It refers to represent necessary features without including more details or explaination. Data abstraction is a programming technique that relies on the separation of interface and implementation.
  - Take an example of your laptop.
  - When you press a key on keyboard, the character appears on the screen.
  - You need to know only this.
  - How exactly it works is not required. This is called abstraction.

OOP Principles ?

- **Inheritance**: The mechanism of deriving a new class from an old class is called inheritance. The old class is known as base class while new class is known as derived class or subclass. The inheritance is the most powerful features of OOP.
  - For example, a child inherits the traits of his/her parents.
  - With inheritance, we can reuse the fields and methods of the existing class.
  - Through effective use of inheritance, you can save lot of time in your programming and also reduce errors
  - Which in turn will increase the quality of work and productivity.
  - There are different types of inheritance: single, multiple, multilevel, hybrid.

OOP Principles ?

- **Polymorphism**: It means ability to take more than one form.
  - For example, a + is used to add two numbers, but it can also be used to concatenate two strings.
  - This is known as operator overloading because same operator may behave differently on different instances.
  - Same way functions can be overloaded. For eg, sum() can be used to add two integers as well as two floating point numbers.

Many

Name

@

2 + 3 addition = 5

"abc" + "def" = abcdef

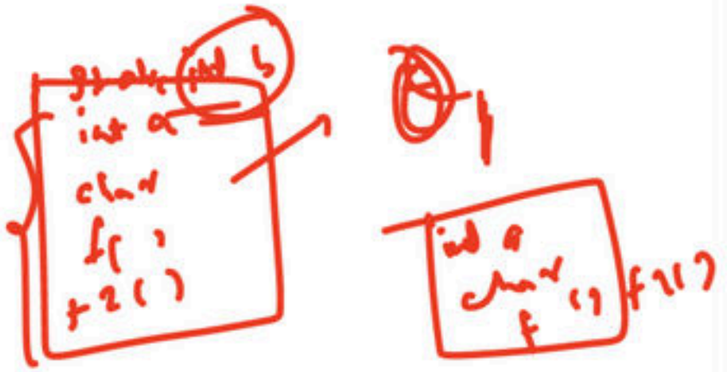
Object Oriented programming in C++

Classes, Objects & Methods

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

C++ Classes and Objects



- C++ is an object-oriented programming language.

- Everything in C++ is associated with classes and objects, along with its attributes and methods.

- Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members".

- A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "**blueprint**" for creating objects.

Creating a class

To create a class, use the **class** keyword:

For eg:

```
class MyClass {     // Class name
 public:                // Access specifier
   int myNum;       // Attribute
   string myString;  // Attribute
};
```


variables - ability
functions - methods

Creating an Object

In C++, an object is created from a class.

To create an object, specify the class name, followed by the object name.

To access the class attributes, use the dot syntax (.) on the object:

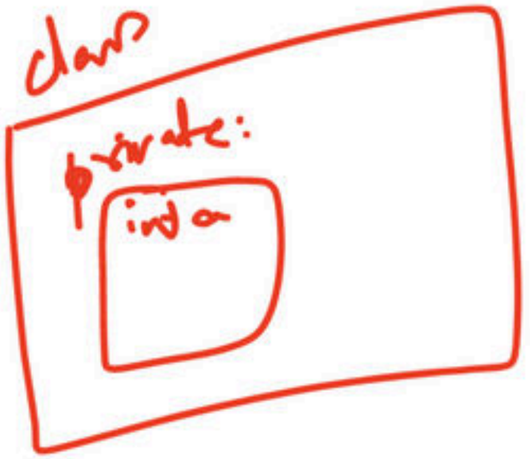
```
MyClass myObj;  // Create an object of MyClass

myObj.myNum = 5;

myObj.myString = "Hello";
```

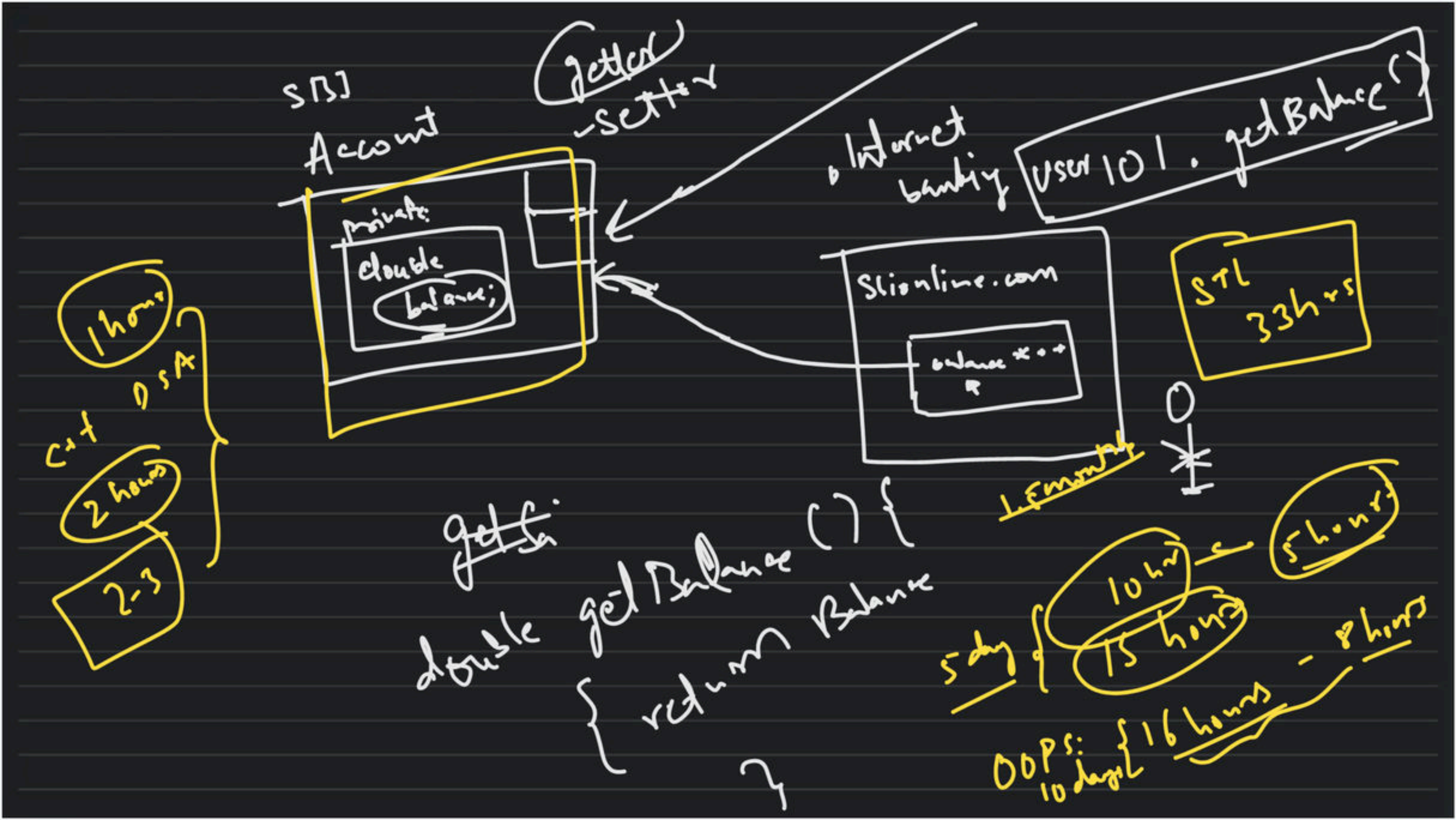
Class Methods

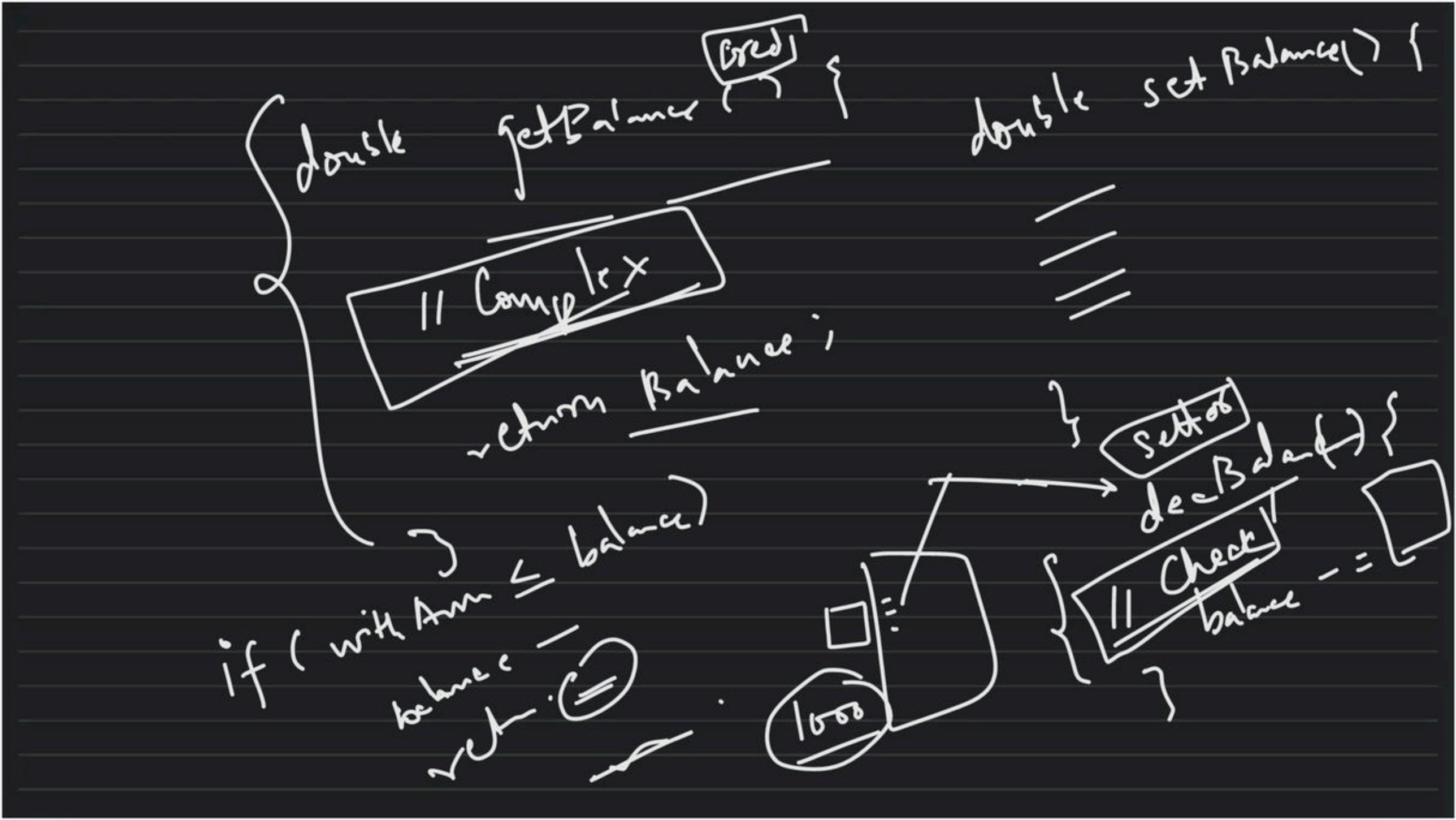
Methods are functions that belongs to the class. They define the behaviour/ action taken by object.

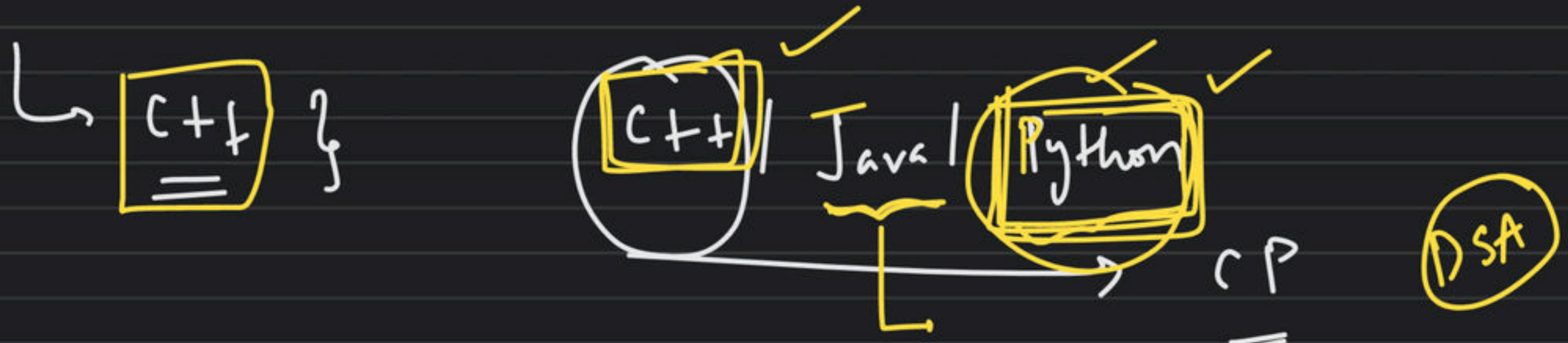
There are two ways to define functions that belongs to a class:

```
class MyClass {          // Class
 public:                 // Access specifier
  void myMethod() {      // Method
   cout << "Hello World!";
  }
};
```




STL
Account
Getter
-setter
private:
double
balance;
Internet
banking
User 101 . getBalance()
Sbionline.com
balance ***
STL
33hrs
1hour
DSA
C++
2 hours
2-3
getBalance
double getBalance () {
return Balance
}
Encapsulation
5 day {
10 hrs
15 hours
5 hours
OOPs:
10 days { 16 hours - 8 hours


double getBalance() {
// Complex
return balance;
}
double setBalance() {
setter
decBalance() {
// Check
balance -=
if (withAmt ≤ balance)
balance -=
1000


C++
C++
Java
Python
CP
DSA

Class Methods

Methods are functions that belongs to the class. They define the behaviour/ action taken by object.

There are two ways to define functions that belongs to a class:

```
class MyClass {          // Class
  public:                // Access specifier
    void myMethod();     // Method declaration
};

void MyClass::myMethod() {
  cout << "Hello World!";
}
```

Class Methods

Eg.

```
class Student {
public:
// Declaration of state/Properties.
   string name;       // Instance variable.
   int rollNo;        // Instance variable.
   static int age;    // Static variable.
// Declaration of Actions.
   void display() { // Instance method.
     // method body.
   }
};
```

Real life example of classes & objects

Object: **Person**

State/Properties:

Black Hair Color: hairColor = "Black";

5.5 feet tall: height = 5.5;

Weighs 60 kgs: weight = 60;

Behaviour/Action:

Eat: eat()

Sleep: sleep(5)

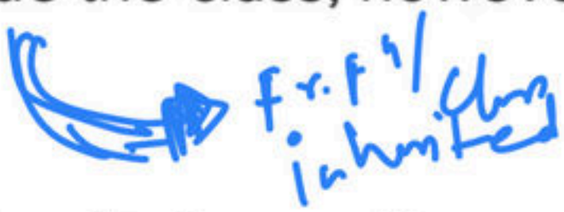
Run: run(0.5)

Access specifiers

Access specifiers define how the members (attributes and methods) of a class can be accessed:

In C++, there are three access specifiers:

- **public** - members are accessible from outside the class
- **private** - members cannot be accessed (or viewed) from outside the class
- **protected** - members cannot be accessed from outside the class, however, they can be accessed in **inherited** classes.



By default, all members of a class are private if you don't specify an access specifier, they will be private

Access specifiers

```
class MyClass {
 public:   // Public access specifier
   int x;  // Public attribute
 private:  // Private access specifier
   int y;  // Private attribute
};

int main() {
 MyClass myObj;
 myObj.x = 5;  // Allowed (public)
 myObj.y = 5;  // Not allowed (private)
 return 0;
}
```

error: y is private

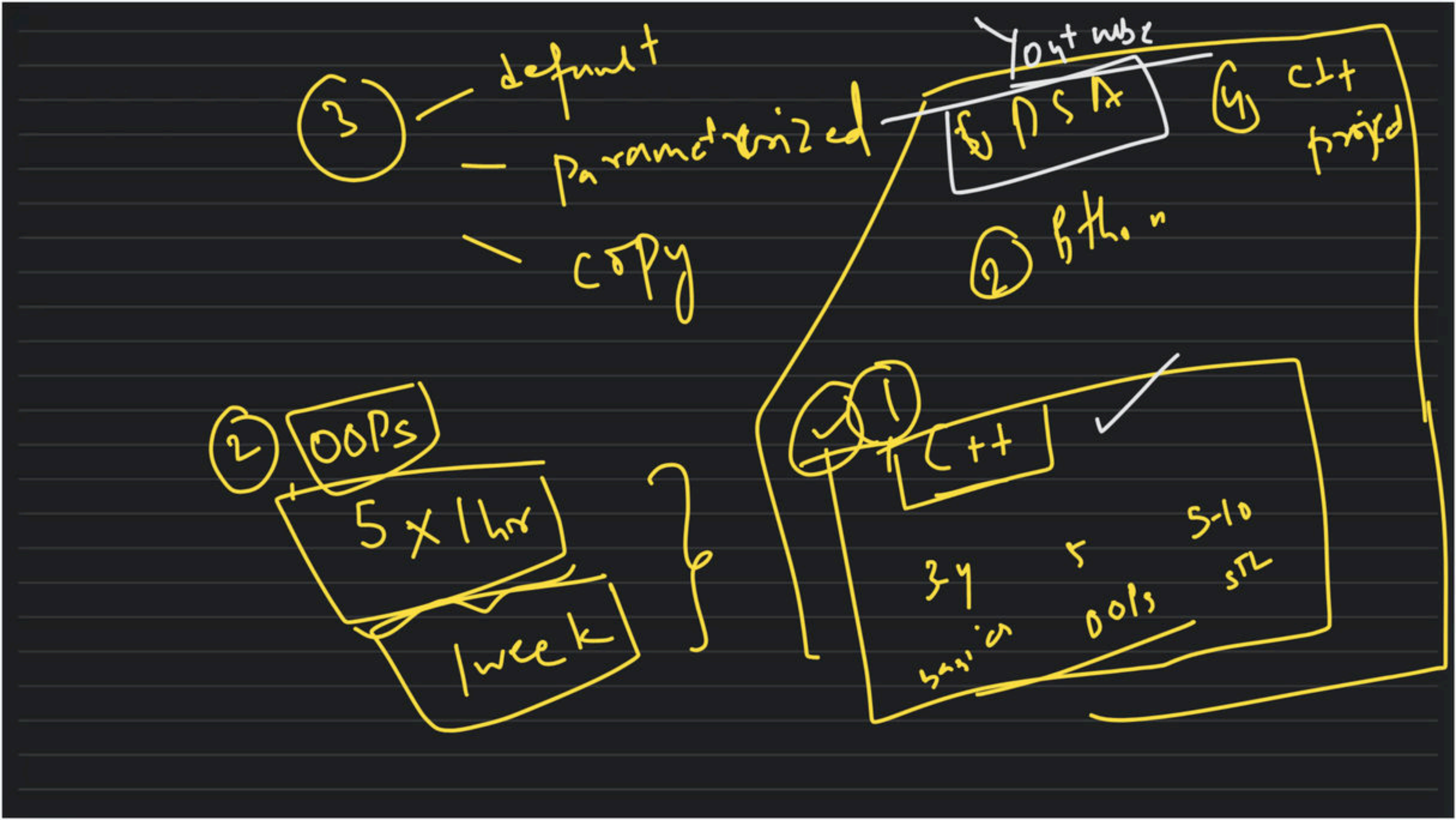
Default Copying

By default, objects can be copied. In particular, a class object can be initialized with a copy of an object of its class. For example:

```
Date d1 = my_birthday;   // initialization by copy

Date d2 {my_birthday};   // initialization by copy
```

By default, the copy of a class object is a copy of each member.

If that default is not the behavior wanted for a class X, a more appropriate behavior can be provided using overloading.


3
default
Parameterized
copy
Youtube
DSA
C++ project
2 Python
1
C++
3y
5
5-10
basics
OOPs
STL
2
OOPs
5 x 1hr
1 week

Interview + CP sheet

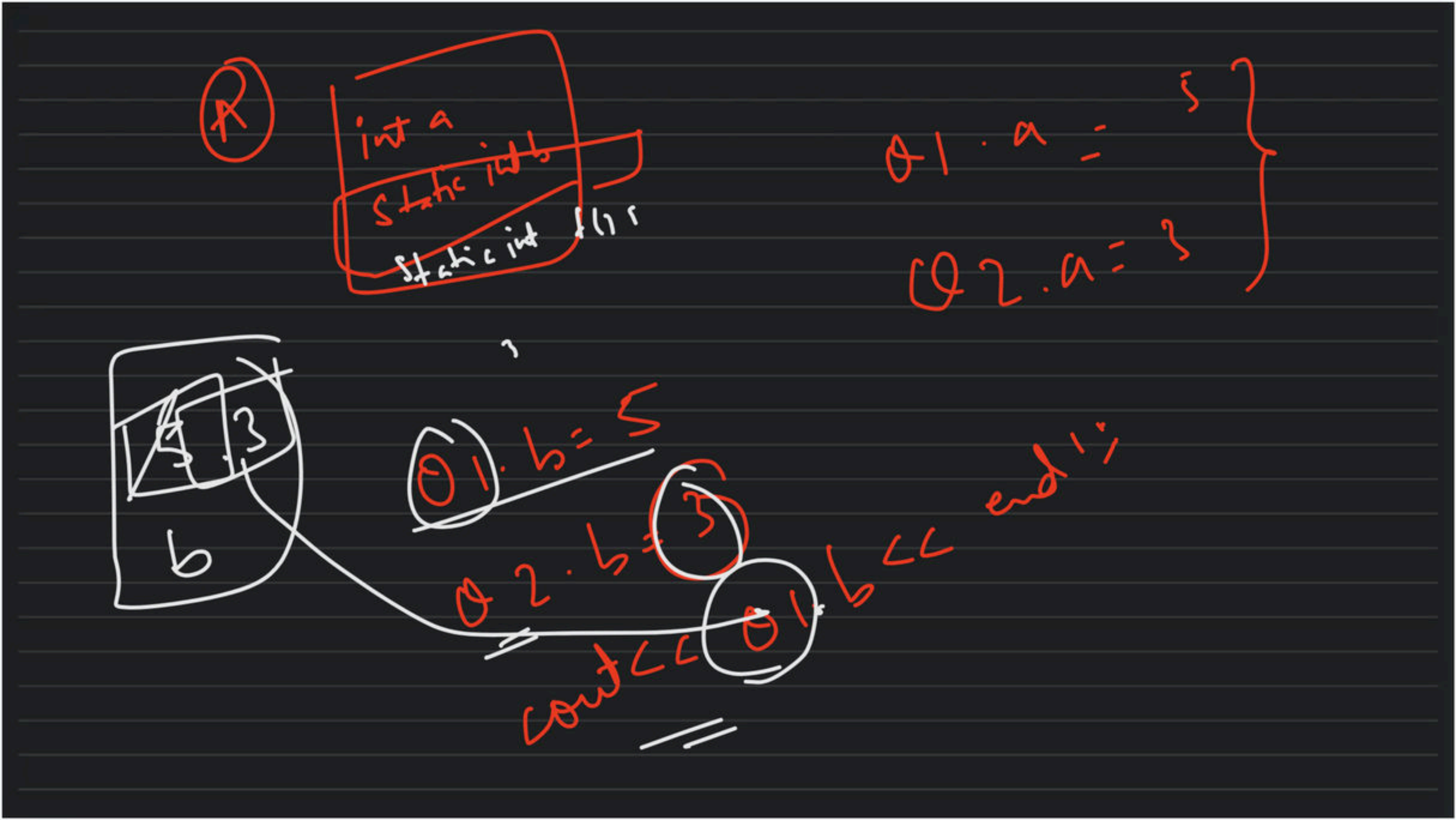
Static Members of a C++ Class

We can define class members static using static keyword.

When we declare a member of a class as static it means no matter how many objects of the class are created, **there is only one copy of the static member.**

A static member is shared by all objects of the class.

All static data is **initialized to zero** when the first object is created, if no other initialization is present.


R
int a
static int b
static int fun()
o1.a = 5
o2.a = 3
b
5
3
o1.b = 5
o2.b = 5
cout << o1.b << endl;

Static Members of a C++ Class

We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Eg:

```
class Box {
  public:
    static int objectCount;
};
// Initialize static member of class Box
int Box::objectCount = 0;
```

Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class.

A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator **::**

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class.

Static Function Members

```
class Box {
  public:
       static int objectCount;
       static int getCount() {
        return length;
    }
  private:
    double length;    // Length of a box
};


Box::getCount();
```

Static Function Members

```
class Box {
  public:
       static int objectCount;
       static int getCount() {
        return objectCount;
    }
  private:
    double length;    // Length of a box
};


Box::getCount();
```

Passing and Returning Objects in C++

In C++ we can pass class objects as arguments and also return them from a function the same way we pass and return other variables.

Passing an Object as argument

To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

Syntax:

```
fun(object_name);
```

Passing and Returning Objects in C++

Passing an Object as argument

```
class Example {
public:
   int a;
   void add(Example E)
   {
      a = a + E.a;
   }
};
```

```
Example E1, E2;
E1.a = 5;
E2.a = 10;
E2.add(E1);
cout << E1.a << E2.a << "\n";
```

Passing and Returning Objects in C++

Returning Object as argument

```
class Example {
public:
   int a;
Example add(Example Ea, Example Eb) {
      Example Ec;
      Ec.a = Ec.a + Ea.a + Eb.a;
      return Ec;
   }
};
```

```
Example E1, E2,E3;
E1.a = 5;
E2.a = 10;
E3 = E3.add(E1, E2);
cout << E3.a << "\n";
```

Friend Function and classes

A **friend class** can access private and protected members of other class in which it is declared as friend

```
class A {
private:
   int a;

public:
   A() {
        a = 0;
   }
   friend class B; // Friend Class
};
```

```
class B {
private:
   int b;
public:
   void showA(A& x) {
      cout << x.a;
   }
};
```

Friend functions and classes

```
class Example {
public:
   int a;
   void add(Example E)
   {
      a = a + E.a;
   }
};
```

```
Example E1, E2;
E1.a = 50;
E2.a = 100;
E2.add(E1);
cout << E1.a << E2.a << "\n";
```

Friend Function and classes

A **friend function** can also be given special grant to access private and protected members. A friend function can be:

a) A method of another class

b) A global function

```
class A {
private:
   int a;
public:
   friend void B::showA(A& x);
};
```

```
class B {
private:
   int b;
public:
   void showA(A& x) {
      cout << x.a;
   }
};
```

Const member functions in C++

An object declared as const cannot be modified and hence, can invoke only const member functions as these functions ensure not to modify the object.

A const object can be created by prefixing the const keyword to the object declaration.

Any attempt to change the data member of const objects results in a compile-time error.

Whenever an object is declared as const, it needs to be initialized at the time of declaration. however, the object initialization while declaration is possible only with the help of constructors.

eg.

```
const clA oba;
```

Const member functions in C++

- When a function is declared as const, it can be called on any type of object, const object as well as non-const objects.

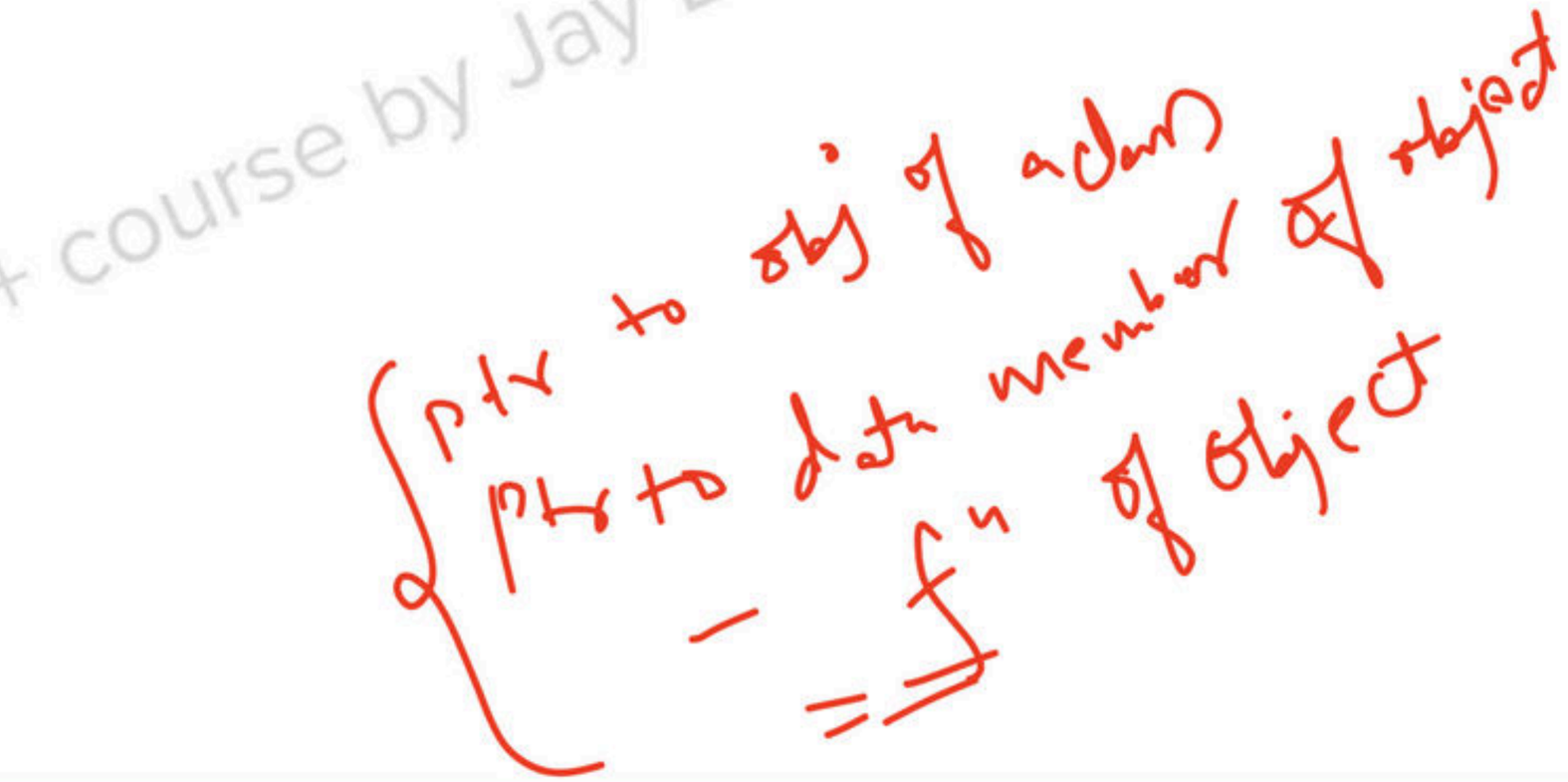
- const functions are not allowed to modify the object on which they are called

Pointers to class members

Defining a Pointer of Class type

We can define pointer of class type, which can be used to point to class objects:

```
Clss obj;
Clss* ptr;
ptr = &obj;
cout << obj.a;
cout << ptr -> a;
```

Pointers to class members

Pointer to Data Members of Class

We can use pointer to point to class's data members (Member variables):

Syntax for Declaration :

datatype class_name :: *pointer_name;

Syntax for Assignment:

pointer_name = &class_name :: datamember_name;

Pointers to class members

Pointer to Data Members of Class

```
class Data {
   public:
   int a;
   void print() {
      cout << "a is "<< a;
   }
};
```

```
Data d, *dp;
dp = &d;    // pointer to object
int Data::*ptr=&Data::a;
//pointer to data member 'a'

   d.*ptr=10;
   d.print();

   dp->*ptr=20;
   dp->print();
```

Pointers to class members

Pointer to Member Functions of Class

Pointers can be used to point to class's Member functions.

Syntax:

return_type (class_name::*ptr_name) (argument_type) = &class_name::function_name;

Pointers to class members

Pointer to Member Functions of Class

```
class Data {
   public:
   int f(float) {
      return 1;
   }
};

int (Data::*fp1) (float) = &Data::f;
```

Object Oriented programming in C++

Constructors & Destructors

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

C++ Constructors

- A constructor in C++ is a special method of class that is automatically called when an object of a class is created

- It is used to initialise objects' attributes

- Constructors have same name as that of the class

- These are must to make public

- Constructors do not have any return value

C++ Constructors

- Constructors can also take parameters (just like regular functions), which can be useful for setting initial values for attributes

- As with functions, constructors are also allowed to define outside the class

- If we don't define a constructor in the class, C++ compiler generates a default constructor for us which takes no parameter as input and has an empty body.

Constructors syntax

```
class myClass{
    public:
        myClass(){
            //constructor
        }
};
```

Types of Constructors

Constructors are broadly classified in three types:

- Default Constructor
- Parameterized Constructor
- Copy Constructor

Default Constructor

- Default constructor is the one having an empty parameter list
- Syntax:

```
class A{
public:
    A(){
        //Default constructor
    }
};
```

- If we don't define it explicitly, compiler creates an empty default constructor as an **inline public member** of its class.

Parameterized Constructor

- Constructor with parameters is called Parameterized constructors
- Syntax:

```
class A(){
    int a, b;
    public:
        A(int x, int y){
        a = x; b =y;
        }
};
```

- Parameterized constructors are preferred to initialize data members

Copy Constructor

The copy constructor in C++ is used to copy data of one object to another.

```
class Example {
public:
    Example( const Example & ); // Declare copy constructor.
    // ...
};

int main() {
}
```

Destructors

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

Syntax: **~destructor_name();**

- Destructor function is automatically invoked when the objects are destroyed.
- The destructor does not have arguments, no return type not even void
- It cannot be declared static or const.
- A destructor should be declared in the public section of the class.
- You cannot access the address of destructor.
- A destructor function is called automatically when the object goes out of scope
- There can only one destructor in a class

Order of Destruction

When an object goes out of scope or is deleted, the sequence of events in its complete destruction is as follows:

- The class's destructor is called, and the body of the destructor function is executed.

- Destructors for nonstatic member objects are called in the reverse order in which they appear in the class declaration. The optional member initialization list used in construction of these members does not affect the order of construction or destruction.

Object Oriented programming in C++

Inheritance

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

C++ Inheritance

- In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.

- In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

- In C++, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base class**. The derived class is the **specialized class** for the base class.

Derived Classes

A Derived class is defined as the class derived from the base class:

The Syntax of Derived class:

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

derived_class_name: It is the name of the derived class.

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

Public, Protected and Private inheritance

- public inheritance makes public members of the base class public in the derived class, and the protected members of the base class remain protected in the derived class.

- protected inheritance makes the public and protected members of the base class protected in the derived class.

- private inheritance makes the public and protected members of the base class private in the derived class.

Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	NOT inherited	NOT inherited	NOT inherited

public < protected < private

Types of Inheritance

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

Single Inheritance

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

```
class Base {
  public:
  float salary = 50000;
};
class Derived: public Base {
  public:
  float bonus = 2000;
};
```



```
int main(void) {
    Derived p1;
     cout<<"Salary: "<<p1.salary<<endl;
     cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

```
Output:
Salary: 50000
Bonus: 2000
```

Single Inheritance

```
class Base {
  public:
  int mul() {
       return a*b;
   }
};
class Derived: private Base {
  public:
  void f()  {
       return mul(a*b);
   }
};
```

```
int main(void) {
   Derived b;
   cout << b.f(2, 3) << endl;
   return 0;
}
```

```
Output:
6
```

Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

```
class Base {
  public:
  float salary = 50000;
};
class Derived1: public Base {
};
class Derived2: public Base {
  public:
  float bonus = 2000;
};
int main(void) {
    Derived2 p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

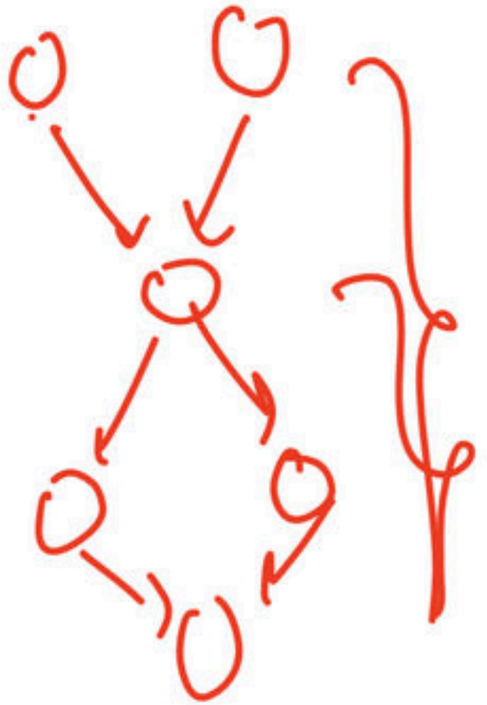
```
Output:
Salary: 50000
Bonus: 2000
```

Multiple Inheritance

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

Syntax of the Derived class:

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

Ambiguity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base classes.

```
class A {
    public:
    void fun() {
        cout << "Class A" << endl;
    }
};
class B {
    public:
    void fun() {
        cout << "Class B" << endl;
    }
};
```

```
class C : public A, public B {
    public:
    void fun2() {
        fun();
    }
};
```

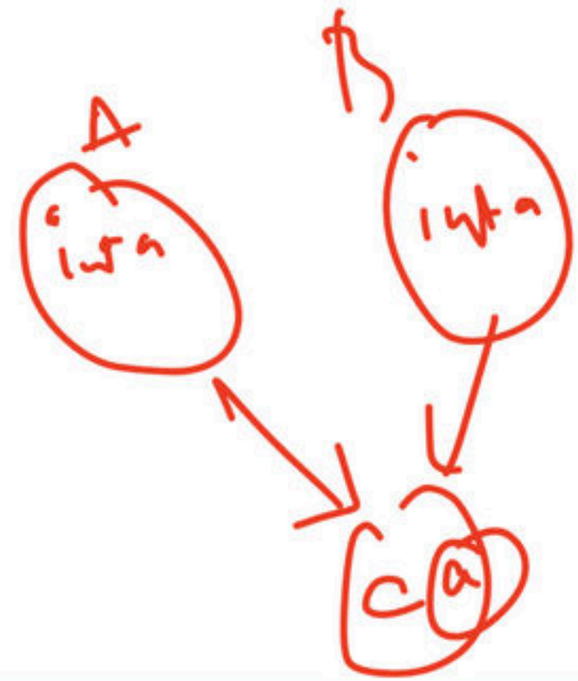
```
error: reference to 'display' is ambiguous
        display();
```

Ambiguity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base classes.

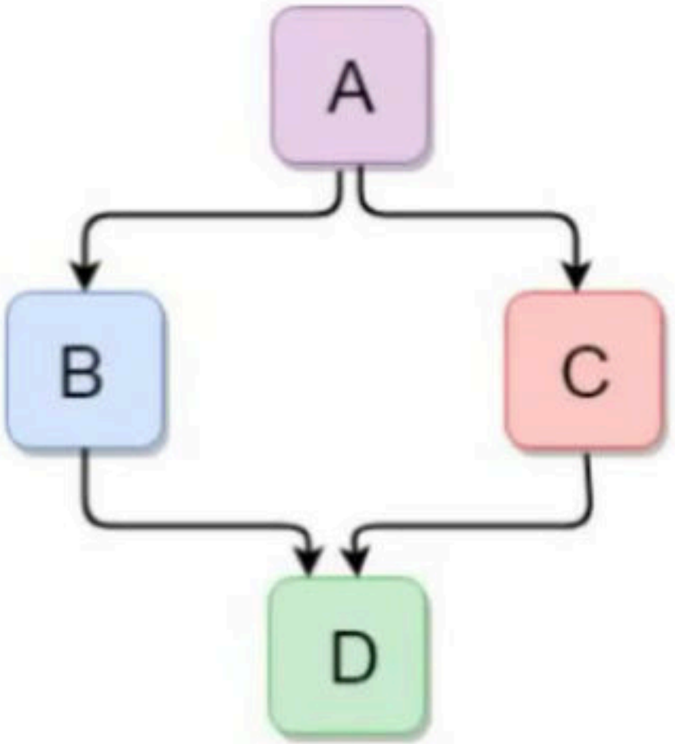
```
class A {
    public:
    void fun() {
        cout << "Class A" << endl;
    }
};
class B {
    public:
    void fun() {
        cout << "Class B" << endl;
    }
};
```

```
class C : public A, public B {
    public:
    void fun2() {
        A :: fun();
        B :: fun();
    }
};
```

Hybrid Inheritance

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



Object Oriented programming in C++

Data Hiding, Encapsulation, Abstraction, Polymorphism

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Data Hiding

- Data hiding is a concept in object-oriented programming which confirms the security of members of a class from unauthorized access

- Data hiding is a technique of protecting the data members from being manipulated or hacked from any other source.

- Data hiding also reduces some complexity of the system. Data hiding can be achieved through the encapsulation, as encapsulation is a subprocess of data hiding.

- Data hiding guarantees restricted data access to class members & maintain object integrity.

Data Hiding

private, public & protected are three types of protection/ access specifiers available within a class. Usually, the data within a class is private & the functions are public. The data is hidden, so that it will be safe from accidental manipulation.

- Private members/methods can only be accessed by methods defined as part of the class. Data is most often defined as private to prevent direct outside access from other classes. Private members can be accessed by members of the class.
- Public members/methods can be accessed from anywhere in the program. Class methods are usually public which is used to manipulate the data present in the class. As a general rule, data should not be declared public. Public members can be accessed by members and objects of the class.
- Protected member/methods are private within a class and are available for private access in the derived class.

Encapsulation

- Encapsulation binds the data & functions together which keeps both safe from outside interference. Data encapsulation leads to data hiding.

- The private members of a class are only accessible to the objects of that class only, and the public members are accessible to the objects of the class as well as they are accessible from outside the class. Encapsulation helps the end user of a system to learn what to do with the system instead of how it must do.

- Encapsulation makes the system easier to operate by the end user.

Abstraction

- Abstraction is primarily used to hide the complexity.

- It indicates the necessary characteristics of an object that differentiates it from all other types of objects.

- An abstraction concentrates on the external aspect of an object. For an object, abstraction provides the separation of the crucial behaviour from its implementation.

- A proper abstraction emphasizes on the details that are important for the reader or user and suppresses features that are, irrelevant and deviant

Types of Abstraction

- Procedural abstraction – It includes series of the instructions having the specified functions. Procedural abstraction provides mechanisms for abstracting well defined procedures or operations as entities. The implementation of the procedure requires a number of steps to be performed.

- Data abstraction – It is set of data that specifies and describes a data object. This principle is at the core of Object Orientation. In this form of abstraction, instead of just focusing on operations, we focus on data first and then the operations that manipulate the data.

Data Hiding v/s Encapsulation v/s Abstraction

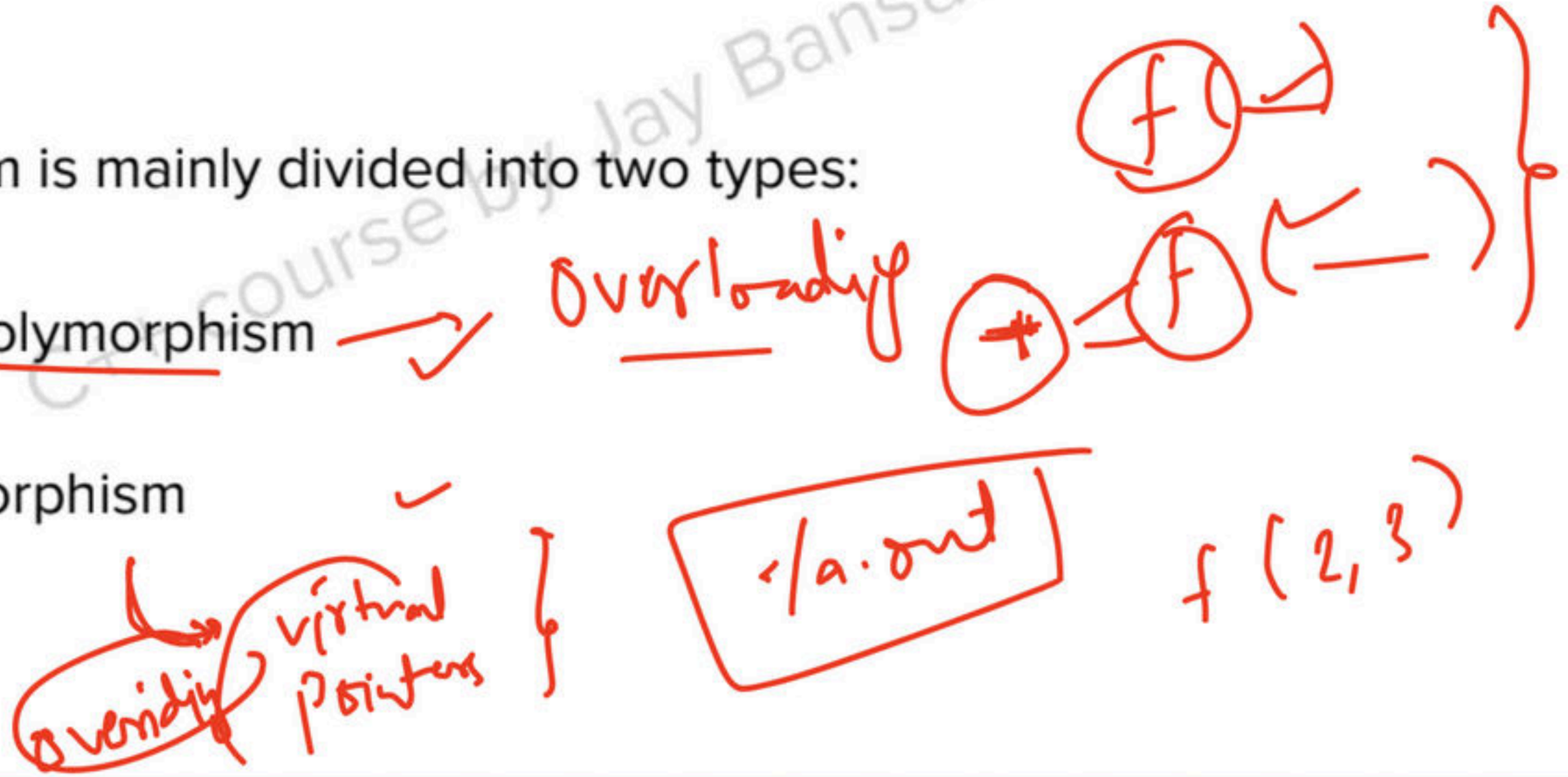
Basis	Data Hiding	Encapsulation	Abstraction
Definition	Hides the data from the parts of the program	Encapsulation concerns about wrapping data to hide the complexity of a system	Extracts only relevant information and ignore inessential details
Purpose	Restricting or permitting the use of data inside the capsule	Enveloping or wrapping the complex data	Hide the complexity
Access Specifier	The data under data hiding is always private and inaccessible	The data under encapsulation may be private or public	-

Polymorphism

- The word **polymorphism** means having many forms. We can define polymorphism as the ability of a message to be displayed in more than one form.

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

- Compile time Polymorphism
- Runtime Polymorphism



Compile time Polymorphism

This type of polymorphism is achieved by function overloading and operator overloading.

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

Rules for function overloading

- Function declarations that differ only in the return type are equivalent
- Parameter declarations that differ only in a pointer * versus an array [] are equivalent
- Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent
- Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent
- Two parameter declarations that differ only in their default arguments are equivalent

Compile time Polymorphism

This type of polymorphism is achieved by function overloading and operator overloading.

- **Operator Overloading**:
  C++ also provide option to overload operators.
  Eg. 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

Run-time Polymorphism

Runtime polymorphism is also known as dynamic polymorphism or late binding. In runtime polymorphism, the function call is resolved at run time.

In contrast, to compile time or static polymorphism, the compiler deduces the object at run time and then decides which function call to bind to the object
This type of polymorphism is achieved by Function Overriding.

- **Function Overriding**:
  Function overriding 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.

Object Oriented programming in C++

Operator Overloading

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Operator Overloading

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as operator overloading

Examples:

Syntax of Operator Overloading

To overload an operator, we use a special operator function:

```
class className {
    public
        returnType operator symbol (arguments) {
            ...
        }
};
```


returnType is the return type of the function.

operator is a keyword.

symbol is the operator we want to overload. Like: +, <, -, ++, etc.

arguments is the arguments passed to the function

Operator Overloading in Unary Operators

Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

```
class Count {
    public:
     int value;
     void operator ++ () {
          ++value;
     }
     void operator ++ (int){
          ++value;
     }
};
```

```
Count count1;
count1.value = 3;
count1++;
cout << count1.value;
++ count1;
cout << count1.value;
```

Operator Overloading in Binary Operators

Binary operators work on two operands.

```
class Complex {
    public:
     int r,i;
     Complex operator + (const Complex& obj){
          Complex temp;
          temp.r = r + obj.i;
          temp.i = r + obj.i;
          return temp;
     }
};
```

Operator Overloading

- Two operators = and **&** are already overloaded by default in C++. For example, to copy objects of the same class, we can directly use the = operator. We do not need to create an operator function.

- Operator overloading cannot change the precedence and associativity of operators.

- There are 4 operators that cannot be overloaded in C++. They are:

:: (scope resolution)

. (member selection)

.* (member selection through pointer to function)

?: (ternary operator)

Different approaches to Operator Overloading

You can perform operator overloading by implementing any of the following types of functions:

- Member Function
- Non-Member Function
- Friend Function

The operator overloading function may be a member function when a Left operand is an object of the Class.
When the Left operand is different, the Operator overloading function should be a non-member function.
You can make the operator overloading function a friend function if it needs to access the private and protected class members.

Restrictions to Operator Overloading

- The operators :: (scope resolution), . (member access), .* (member access through pointer to member), and ?: (ternary conditional) cannot be overloaded.

- New operators such as **, <>, or &| cannot be created.

- It is not possible to change the precedence, grouping, or number of operands of operators.

- The overload of operator -> must either return a raw pointer, or return an object (by reference or by value) for which operator -> is in turn overloaded.

- The overloads of operators && and || lose short-circuit evaluation.

Function call operator

When a user-defined class overloads the function call operator, operator(), it becomes a FunctionObject type.

An object of such a type can be used in a function-call-like expression:

```
struct Linear {
   double a, b;

   double operator()(double x) const{
        return a*x + b;
    }
};
```

```
Linear f{2, 1};
 Linear g{-1, 0};

double f_0 = f(0);
double f_1 = f(1);

double g_0 = g(0);
```

Object Oriented programming in C++

Virtual Class, Virtual Methods, Abstract Classes

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Runtime Polymorphism

- Runtime polymorphism is also known as dynamic polymorphism or late binding. In runtime polymorphism, the function call is resolved at run time.

- In contrast, to compile time or static polymorphism, the compiler deduces the object at run time and then decides which function call to bind to the object. In C++, runtime polymorphism is implemented using **method overriding**.

Method Overriding

- Inheritance is a feature of OOP that allows us to create derived classes from a base class. The derived classes inherit features of the base class.

- If the same function is defined in both the derived class and the based class and we call this function using the object of the derived class, the function of the derived class is executed.

- This is known as function overriding in C++. The function in derived class overrides the function in base class.

Method Overriding

```
class B {
  public:
   void print() {
      cout << "Base Function" << endl;
   }
};

class D : public B {
  public:
   void print() {
      cout << "Derived Function" << endl;
   }
};
```

```
D d1, d2;
d1.print();
d2.B::print();
```

```
Output:
Derived Function
Base Function
```

Method Overriding

```
class B {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class D : public B {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};
```

```
D d1;
B* ptr = &d1;
ptr->print();
```

```
Output:
Base Function
```

Virtual Functions

- A virtual function is a member function in the base class that we **expect to redefine** in derived classes.

- A virtual function is used in the base class in order to ensure that the function is overridden. This especially applies to cases where a pointer of base class points to an object of a derived class.

Virtual Functions

```
class B {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class D : public B {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};
```

```
D d1;
B* b1 = &d1;
b1->print();
```

```
Output:
Base Function
```

Virtual Functions

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

class D : public B {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};
```

```
D d1;
B* b1 = &d1;
b1->print();
```

```
Output:
Derived Function
```

override Identifier

- override identifier specifies the member functions of the derived classes that override the member function of the base class.

- Using the override identifier prompts the compiler to display error messages when these mistakes are made, Otherwise, the program will simply compile but the virtual function will not be overridden.

- Errors handled: incorrect names, different return type, different parameters, No virtual function.

Example

```
class B {
    int a;
  public:
    virtual int fun() { return a; }
};
class D1 : public B {
    int a;
  public:
    int fun() override { return a; }
};
class D2 : public B {
    int a;
  public:
    int fun() override { return a; }
};
```

```
void print(B* b) {
    cout << "Value: " << b->fun() << endl;
}

B* b1 = new B(1);
B* b2 = new D1(2);
B* b3 = new D2(3);
print(b1);
print(b2);
print(b3);
```

```
Output:
1
2
3
```

Virtual base classes

- Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances

Example:

Pure virtual functions & Abstract Classes

- Pure virtual functions are used if a function **doesn't have** any use in the base class but the function **must** be implemented by all its derived classes

- A class that contains a pure virtual function is known as an abstract class.

- We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions)

Pure virtual functions & Abstract Classes

```
class B {
    int a;
  public:
    virtual int fun() = 0;
};
class D1 : public B {
    int a;
  public:
    int fun() override { return a; }
};
class D2 : public B {
    int a;
  public:
    int fun() override { return a; }
};
```

```
void print(B* b) {
    cout << "Value: " << b->fun() << endl;
}

B* b1 = new B(1);
B* b2 = new D1(2);
B* b3 = new D2(3);
print(b2);
print(b3);
```

```
Output:
2
3
```

Polymorphism

Object Oriented programming in C++

Remaining Topics in OOPs

Ravindrababu Ravula
Jay Bansal

unacademy.com/@ravula
unacademy.com/@jay.bansal

Virtual Table (VTABLE)

- The compiler at compile time sets up one VTABLE each for a class having virtual functions as well as the classes that are derived from classes having virtual functions.

- A VTABLE contains entries that are function pointers to the virtual functions that can be called by the objects of the class. There is one function pointer entry for each virtual function.

- In the case of pure virtual functions, this entry is NULL.

VTABLE and vptr

- _vptr which is called the vtable pointer is a hidden pointer that the compiler adds to the base class. This _vptr points to the vtable of the class. All the classes derived from this base class inherit the _vptr.

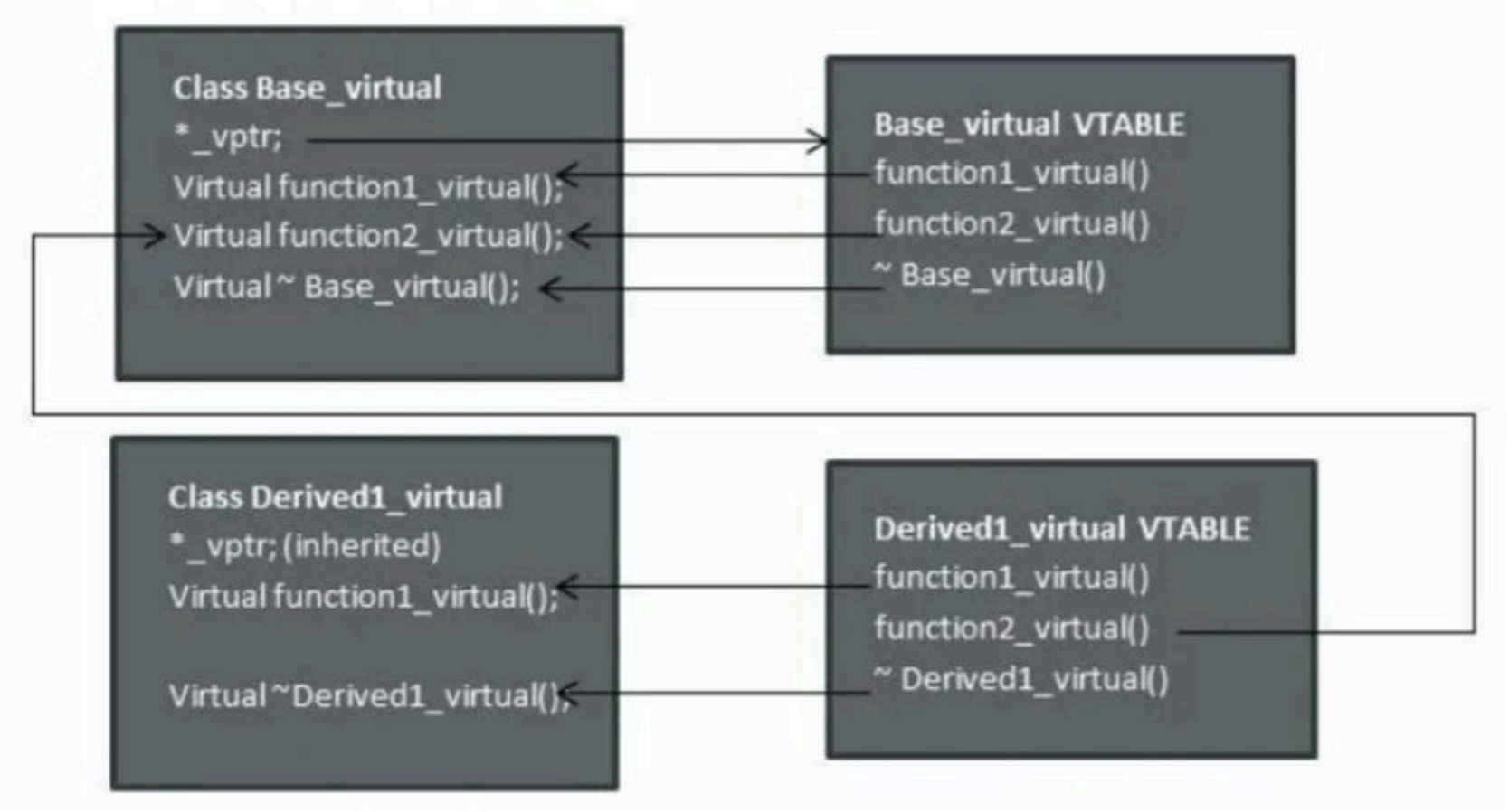
- Every object of a class containing the virtual functions internally stores this _vptr and is transparent to the user. Every call to virtual function using an object is then resolved using this _vptr

Example

```
class Base_virtual{
 public:
 virtual void f1_v() {cout<<"Base :: f1_v()";};
 virtual void f2_v() {cout<<"Base :: f2_v()";};
 virtual ~Base_virtual(){};
};

class Derived1_virtual: public Base_virtual {
public:
 ~Derived1_virtual(){};
 virtual void f1_v() { cout<<"Derived1_virtual :: f1_v()";}
};
```

Example

Now let us see how the above program is represented internally using vtable and _vptr.



Virtual Destructors

```
class Base {
    public:
    ~Base() {
        cout << "Base Class:: Destructor\n";
    }
};
class Derived:public Base {
    public:
    ~Derived() {
        cout<< "Derived class:: Destructor\n";
    }
};
```

```
Base* b = new Derived;
delete b;
```

Virtual Destructors

- We have an inherited derived class from the base class. In the main, we assign an object of the derived class to a base class pointer

- Ideally, the destructor that is called when “delete b” is called should have been that of derived class but we can see from the output that destructor of the base class is called as base class pointer points to that

- Due to this, the derived class destructor is not called and the derived class object remains intact thereby resulting in a memory leak. The solution to this is to make base class constructor virtual so that the object pointer points to correct destructor and proper destruction of objects is carried out

Virtual Destructors

```
class Base {
    public:
    ~Base() {
        cout << "Base Class:: Destructor\n";
    }
};
class Derived:public Base {
    public:
    ~Derived() {
        cout<< "Derived class:: Destructor\n";
    }
};
```

```
Base* b = new Derived;
delete b;
```

JAY BANSAL SIR

Expert in Competitive Programming

GATE CSE 2019 - AIR 2

DM

Incoming SDE @ Google
ACM ICPC 2019 - AIR 39
B.Tech Gold Medalist

Subjects Taken:

Competitive Programming
C++
Data Structures
Algorithms
Interview Preparation

unacademy | CODECHEF

New Batches, JOIN NOW

Adroit 2.0

9-Month Career Readiness Batch for Beginners

One-month Course on Graph Algorithms and Interview Problem Solving

Riya, Deepak

Resolve to become an Expert level Programmer in 2021 and Subscribe at an expense even lesser than **INR 90/day**

Join Our EXCLUSIVE BATCHES

ADROIT 2.0

9 MONTHS COURSE

9-Month Career Readiness Batch for Beginners

LIVE ON **8 JUNE 2021**

One-month Course on Graph Algorithms and Interview Problem Solving

LIVE ON **8 JUNE 2021**

VICTOR

6 MONTHS COURSE

Concise Course for Intermediate Programmers

LIVE ON **31 MAY 2021**

STRIKER

9 MONTHS COURSE

Data Structures Fundamentals with Curated Problem Solving

LIVE ON **31 MAY 2021**

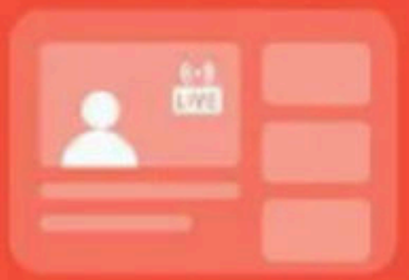
EVEREST 3.0

11 MONTHS COURSE

Complete Course on Programming

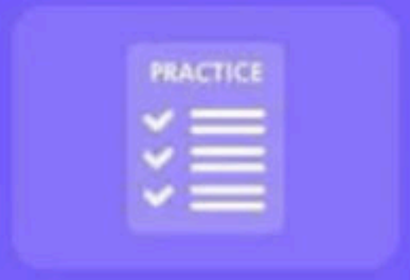
LIVE ON **1 JUNE 2021**

What You Will Get?



Live Interactive Classes

Attend live interactive classes with our top educators. Interact during class with educators to get all your doubts resolved.



Practice Relevant Problems @ Codechef

Each class comes with a set of curated practice problems to help you apply the concepts in real time.

Doubt Support

If you get stuck in any problem post class- Get your doubts resolved by our expert panel of teaching assistants and community members instantly.

What You Will Get ?

Live Interactive Classes

Attend live interactive classes with our top educators. Interact during class with educators to get all your doubts resolved

Practice Relevant Problems @ CodeChef

Each class comes with a set of curated practice problems to help you apply the concepts in real time.

Doubt Support

If you get stuck in any problem post class- Get your doubts resolved by our expert panel of teaching assistants and community members instantly

Resolve to become an Expert level Programmer in 2021 and
Subscribe at an expense even lesser than
INR 90/day

unacademy | CODECHEF

TOPIC WISE

BATCH STRUCTURE

Adroit 2.0:
9-Month Career Readiness Batch for Beginners
(Launching on 8th June)

TOPIC WISE BATCH STRUCTURE

Adroit 2.0: 9-Month Career Readiness Batch for Beginners

Week 1 - 4

Introduction to Programming in C++

Week 5 - 9

Sorting & Searching

Week 10 - 13

Basic Data Structures - Fundamentals and Interview Specific Problem Solving

Week 14 - 17

Additional Concepts in C++

Week 18 - 20

Greedy Algorithms with Classical Problem Solving

TOPIC WISE BATCH STRUCTURE

Adroit 2.0: 9-Month Career Readiness Batch for Beginners

Week 21 - 24

Advanced Data Structures - Square Root Decomposition & Complex Problems

Week 25 - 26

Number Theory And Interview Questions

Week 27 - 28

Recursion on DP- Concept to Detail

Week 29 - 32

Discrete Mathematics

Week 33 - 37

Graph Algorithm- Fundamentals to Extensive Problem Solving

Week 38 - 40

Segment Trees

TOPIC WISE BATCH STRUCTURE

Adroit 2.0: 9-Month Career Readiness Batch for Beginners

Week 41- 43

Advanced Dynamic Programming

Week 44- 46

Computational Geometry

unacademy | CODECHEF

TOPIC WISE

BATCH STRUCTURE

One-month Course on Graph Algorithms and Interview Problem Solving

(Launching on 8th June)

TOPIC WISE BATCH STRUCTURE

One-month Course on Graph Algorithms and Interview Problem Solving

Week 1 - 2

Graph Algorithms- Tackling Complex Interview Questions

Week 3 - 4

Handpicked/ Previous Interview Problems

Our Educators

ICPC World Finalists, CodeForces Grandmasters, Alumni of Top Product Companies

Meet Our Educators

Tanuj Khattar

ACM ICPC World Finalist - 2017, 2018. Indian IOI Team Trainer 2016-2018. Worked @ Google, Facebook, HFT. Quantum Computing Enthusiast.

Sanket Singh

Software Development Engineer @ LinkedIn | Former SDE @ Interviewbit | Google Summer of Code 2019 @ Harvard University | Former Intern @ISRO

Pulkit Chhabra

Codeforces: 2246 | Codechef: 2416 | Former SDE Intern @CodeNation | Former Intern @HackerRank

Riya Bansal

Software Engineer at Flipkart | Former SDE and Instructor @ InterviewBit | Google Women TechMakers Scholar 2018

Triveni Mahatha

Qualified ICPC 2016 World Final. Won multiple Codechef Long Challenges (India). ICPC Onsite Regionals' Problem setter and Judge. IIT Kanpur.

Arnab Chakraborty

Six Sigma Master Black Belt-IQF(USA),AMT by AIMA,NLP & PMP Trained,ITIL-APMG(UK),Control & Automation-ISA(USA),"Star Python"-Star Cert.(USA)

Meet Our Educators

Himanshu Singh

World Finalist ICPC 2020, Winner Techgig Code Gladiators 2020, Winner TCC '19, 2020 CSE Graduate from IIT BHU, Works at Nutanix

Nishchay Manwani

Hey I am Nishchay Manwani from CSE, IIT Guwahati and I'm a Seven star on Codechef and International Grandmaster on Codeforces.

Utkarsh Gupta

I'm a competitive programmer. I am red on CF and 6* on Codechef. I have been teaching CP on various platforms for approx a year.

Arjun Arul

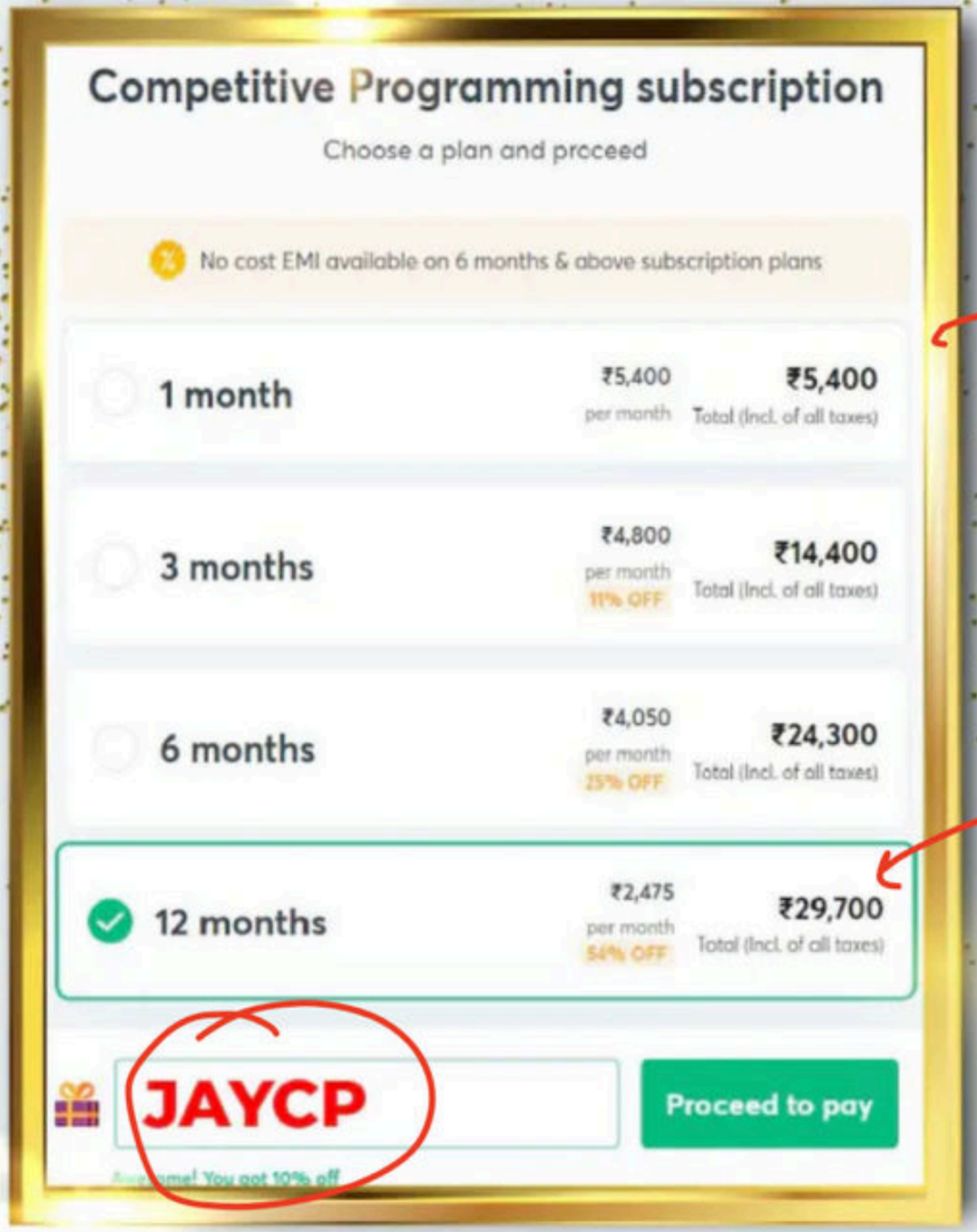
ICPC World Finalist 2012 & 2013, Coach @ IOITC

Deepak Gour

ICPC World Finalist 2020 | Former Instructor @InterviewBit | Software Engineer at AppDynamics

Vivek Chauhan

Codechef: 7 stars (2612) India Rank 6, Codeforces: MASTER (2279), Won Codechef Long Challenges(India), TCO20 Southern Asia Runner up


Competitive Programming subscription
Choose a plan and proceed
No cost EMI available on 6 months & above subscription plans
1 month
₹5,400
per month
₹5,400
Total (Incl. of all taxes)
3 months
₹4,800
per month
11% OFF
₹14,400
Total (Incl. of all taxes)
6 months
₹4,050
per month
25% OFF
₹24,300
Total (Incl. of all taxes)
12 months
₹2,475
per month
54% OFF
₹29,700
Total (Incl. of all taxes)
JAYCP
Proceed to pay
You got 10% off
6k
33 lc

Follow Me on

unacademy

https://unacademy.com/@jay.bansal

Thank You