1. **What are the features of OOPS?**

**Class**: A class is a user-defined datatype. Which has data members and member functions.

**Objects**: An Object is an instance of the class. When a class is defined no memory is allocated but when it is instantiated Memory is allocated.

**Encapsulation**: Binding together the data and functions that manipulate them.

**Role of access specifiers in encapsulation:**

The access specifiers play an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

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

**Abstraction**: Abstraction means displaying only essential information and hiding the details. Data Abstraction refers to providing only essential information to the outside world and hiding the background details or implementation.

**Advantages of Data Abstraction**:

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

**Polymorphism**: Polymorphism means having many forms. we can define polymorphism as the ability of a message to be displayed in more than one form.

An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operations

Polymorphism is divided into two types.

1. **Compile-time polymorphism**
2. **Runtime polymorphism**

Compile time polymorphism is again divided into two types.

1. **Operator Overloading:** The process of making an operator exhibit different behaviors in different instances is known as operator overloading.

**What is the difference between the operator function and the the normal function?**

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

**Can we overload all the operators?**

Almost all operators can be overloaded except a few. Those operators are listed below.

* **Sizeof**
* **Typeid**
* **Scope resolution operator (::)**
* **Class member access operator (‘.’ And ‘->’)**
* **Tenary or conditional operator (?:)**

1. **Function Overloading**: Function overloading is using a single function name to perform different types of tasks.

**Rules Of not Function Overloading:**

* Function declaration differs only in the return type.
* Member functions declaration with the same name and parameter list cannot be overloaded, if anyone of them is a static member function declaration.

**Example:**

**static void fun(int i) {}**

**void fun(int i) {}**

* Pointer declaration that differs only in a pointer versus an array.

**Example:**

**int fun(int \*ptr);**

**int fun(int ptr[]); // redeclaration of fun(int \*ptr)**

* Parameter declaration differs only in that one is a function type, and the other is a pointer to the same function type or equivalent.

**Example:**

**void h(int ());**

**void h(int (\*)()); // redeclaration of h(int())**

* parameter declaration that differs only in const and/or volatile presence are equivalent.

**Example:**

**int f ( int x) {**

**return x+10;**

**}**

**int f ( const int x) {**

**return x+10;**

**}**

Runtime polymorphism is achieved by function overriding.

1. [**Function overriding**](https://www.geeksforgeeks.org/override-keyword-c/): It 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.

**What is the virtual method?**

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined in the derived class.

**What is the use of virtual functions?**

Virtual functions are allowed to create a list of base class pointers to points to derived class objects and call any of the methods in derived classes without knowing the derived class object.

### **How does the compiler perform runtime resolution?**

The compiler maintains two things to serve this purpose.

**Vtable**: A table of function pointers maintained per class.

**Vptr** : A pointer to vtable maintained per object.

# Can Static Functions Be Virtual in C++?

# Static member functions of the class cannot be virtual and those are invoked when you have a pointer or reference to an instance of a class. Static functions aren’t tied to the instance of a class but they are tied to the class. C++ doesn’t have pointers-to-class, so there is no scenario in which you could invoke a static function virtually.

# Pure Virtual Functions and Abstract Classes in C++?

# A pure virtual function is declared by assigning 0 in the declaration.

# 

# Virtual void sample() = 0;how() = 0;

# Some Interesting Facts:

# A class becomes abstract if it has at least one pure virtual function.

# We can have pointers and references for the abstract class type.

# If we do not override the pure virtual function in a derived class, then the derived class also becomes an abstract class.

# An abstract class can have constructors.

# An abstract class in C++ can also be defined using the struct keyword.

# Virtual Destructor?

# Deleting a derived class object using a base class pointer that has a non-virtual destructor then gives the undefined behavior. To correct his situation, the base class should be defined virtual destructor.

# Example :

# #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()

# {

# derived \*d = new derived();

# base \*b = d;

# delete b; // It calls the base destructor only,

# //if we make the base class destructor a virtual destructor

# //then it calls the base & derived class destructors.

# return 0;

# }

# output:

# Constructing base

# Constructing derived

# Destructing base

# if you make line 9 as virtual ~base() then it gives the below output

# output:

# Constructing base

# Constructing derived

# Destructing derived

# Destructing base

**Inheritance**: The capability of a class to derive properties and characteristics from another class is called inheritance.

**The Diamond problem**:

The diamond problem occurs when two base classes of a class have a common base class.

Example:

**#include<iostream>**

**using namespace std;**

**class A**

**{**

**int x;**

**public:**

**void setX(int i) {x = i;}**

**void print() { cout << x; }**

**};**

**class B: public A**

**{**

**public:**

**B() { setX(10); }**

**};**

**class C: public A**

**{**

**public:**

**C() { setX(20); }**

**};**

**class D: public B, public C {**

**};**

**int main()**

**{**

**D d;**

**d.print();**

**return 0;**

**}**

**The above example generates the below print method gets ambiguity to fix this issue derive the class B and class c from the virtual base class**

**#include<iostream>**

**using namespace std;**

**class A**

**{**

**int x;**

**public:**

**A(int i) { x = i; }**

**void print() { cout << x; }**

**};**

**class B: virtual public A**

**{**

**public:**

**B():A(10) { }**

**};**

**class C: virtual public A**

**{**

**public:**

**C():A(10) { }**

**};**

**class D: public B, public C {**

**};**

**int main()**

**{**

**D d;**

**d.print();**

**return 0;**

**}**

**What is object slicing?**

**Object slicing happens when a derived class object is assigned to a base class object**

1. **Access Specifiers**?

There are three types of access specifiers available in c++.

* **Public:**  All the class members declared under the public specifier will be available to everyone. The data members and member functions were declared as public and can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.
* **Private**: The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the [friend functions](https://www.geeksforgeeks.org/friend-class-function-cpp/) are allowed to access the private data of members of the class.
* **Protected:** The protected access modifier is similar to the private access modifier in the sense that it can’t be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

**Note**: This access through inheritance can alter the access modifier of the elements of the base class in the derived class depending on the [mode of Inheritance](https://www.geeksforgeeks.org/inheritance-in-c/#Modes%20of%20Inheritance).

1. **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.

**Example** :

**// 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**

**};**

1. **Difference between class and structure?**

|  |  |
| --- | --- |
| **Class** | **Structure** |
| Members of the class are private by default | Members of structure public by default |
| It is normally used for data abstraction and further inheritance. | It is normally used for the grouping of data |
|  |  |

1. **Static data members in c++?**

* Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
* It is initialized before any object of this class is created, even before the main starts.
* It is visible only within the class, but its lifetime is the entire program

**Some interesting facts about static member functions**

* Static member functions do not have this pointer.

**Example:**

**#include<iostream>**

**class Test {**

**static Test \* fun() {**

**return this; // compiler error**

**}**

**};**

**int main()**

**{**

**getchar();**

**return 0;**

**}**

* Static member functions cannot be virtual.
* Static member functions cannot be declared const, volatile, and const volatile

**Example**:

**#include <iostream>**

**using namespace std;**

**class A**

**{**

**int x;**

**public:**

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

**int getX()**

**{**

**return x;**

**}**

**};**

**class B**

**{**

**static A a;**

**public:**

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

**A getA() { return a; }**

**};**

**// syntax: <<Return-Type>> <<class-name>>::<<static variable/object name>>**

**//static members are defined before execution main.**

**A B::a; // definition of a**

**int main()**

**{**

**B obj;**

**A a = obj.getA();**

**cout<<a.getX()<<endl;**

**return 0;**

**}**

1. **Local classes in c++?**

**A class declared inside the function becomes local to that function and is called a local class in c++.**

* **A local class name can only be used locally i.e., inside the function and not outside it.**
* **The methods of a local class must be defined inside it only.**
* **A local class can have static functions but, not static data members**.
* **Local classes can access global types, variables, and functions.**
* **Member methods of the local class can only access static and enum variables of the enclosing function. Non-static variables of the enclosing function are not accessible inside local classes.**

**Example :**

**#include <iostream>**

**using namespace std;**

**int z = 100;**

**void fun()**

**{**

**static int y = 10;**

**// local to fun**

**class Test**

**{**

**//static int i; // Error : A Local class cannot contain static data members.**

**int x;**

**public:**

**// Fine as the method is defined**

**// inside the local class**

**void method()**

**{**

**//Local class can access global variables and functions**

**//Local class can access only static and enum types from enclosing method but cannot access**

**// the non static variables**

**cout << "Local Class method() called " << x <<" "<<y<< " "<<z<< endl;**

**}**

**//It may contain static functions**

**static void method1()**

**{**

**cout << "Local Class method () called";**

**}**

**//void method1();**

**};**

**// Error as the method is defined outside the local**

**// class**

**/\*void Test::method1()**

**{**

**}\*/**

**Test t; // fine**

**Test \*t1;// fine**

**t.method();**

**}**

**int main()**

**{**

**fun();**

**// Test t; // Error**

**// Test \*t1;// Error**

**return 0;**

**}**

1. **C++ final specifier?**

* Sometimes you don’t want to allow the derived class to override the base class virtual function. [C++ 11](http://en.wikipedia.org/wiki/C%2B%2B11) allows the built-in facility to prevent the overriding of virtual functions using the final specifier.

**Example:**

**#include <iostream>**

**using namespace std;**

**class Base**

**{**

**public:**

**virtual void myfun() final**

**{**

**cout << "myfun() in Base";**

**}**

**};**

**class Derived : public Base**

**{**

**//Error: myfun is final in the base class so we can't override in the derived class**

**void myfun()**

**{**

**cout << "myfun() in Derived\n";**

**}**

**};**

**int main()**

**{**

**Derived d;**

**Base &b = d;**

**b.myfun();**

**return 0;**

**}**

* final specifier in C++ 11 can also be used to prevent inheritance of class/struct. If a class or struct is marked as final then it becomes non-inheritable and it cannot be used as a base class/struct.

**Example:**

**#include <iostream>**

**class Base final**

**{**

**};**

**class Derived : public Base**

**{**

**};**

**int main()**

**{**

**Derived d;**

**return 0;**

**}**

1. **Constructors in c++ ?**

Constructor is a special method that is invoked automatically at the time of object creation.

It is used to initialize the data members of new objects generally.

The constructor in the c++ has the same name as the class or structure.

Constructor is invoked at the time of object creation.

**Characteristics of the constructor**:

* The name of the constructor is the same as its class name.
* Constructors are mostly declared in the public section of the class though they can be declared in the private section of the class.
* Constructors do not return values; hence they do not have a return type.
* A constructor gets called automatically when we create the object of the class.
* Constructors can be overloaded.
* Constructors cannot be declared as virtual.

### 

### **Types of Constructors :**

* **Default constructor**: The default constructor is the constructor which doesn’t take any argument. It has no parameters. It is also called a zero-argument constructor.
* **Parameterized Constructors:**It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.
* **Copy Constructor**:  A copy constructor is a member function that initializes an object using another object of the same class.

**When is a user-defined copy constructor needed?**

We need to define our copy constructor only if an object has pointers or any runtime allocation of the resource like a file handle, a network connection, etc.

**The default constructor only does shallow copy, Deep copy is possible with the user-defined copy constructor**

### 

### **Difference between Copy constructor vs Assignment Operator**

A copy constructor is called when a new object is created from an existing object.

The assignment operator is called when the already initialized object is assigned a new value from another existing object.

**Example :**

**MyClass t1, t2;**

**MyClass t3 = t1; // copy constructor**

**t2 = t1; // Assignment operator**

**Can we make the copy constructor private**?

Yes, A copy constructor can be made private, when we make copy constructor private in a class **objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources.**

**Example of copy constructor**:

#include <iostream>

#include <cstring>

#include <string>

using namespace std;

class copyconstructor

{

char \*s;

int size;

public:

copyconstructor(const char \*strval = nullptr)

{

size = strlen(strval);

s = new char[size+1];

strcpy(s,strval);

}

copyconstructor(const copyconstructor& obj)

{

size = strlen(obj.s);

s = new char[size+1];

strcpy(s, obj.s);

}

~copyconstructor()

{

cout<<"Inside the destructor : "<<endl;

delete []s;

}

void update(const char ch)

{

s[0] = ch;

}

void print()

{

cout<<s<<endl;

}

};

int main()

{

copyconstructor obj("Rajkumar");

obj.print();

copyconstructor obj1 = obj;

obj1.print();

obj.update('v');

obj.print();

obj1.print();

return 0;

}

1. **Destructor**?

* The destructor destroys the class objects created by the constructor.
* Destructor has the same name as their class name preceded by a tiled (~) symbol.
* It is not possible to define more than one destructor.
* The destructor is only one way to destroy the object created by the constructor.
* Hence destructor can-not be overloaded.
* Destructor neither requires any argument nor returns any value.
* It is automatically called when the object goes out of scope.
* Destructors release memory space occupied by the objects created by the constructor.
* In destructor, objects are destroyed in the reverse of object creation.

### **Characteristics of a destructor:-**

* The destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by the program.
* Destructor neither requires any argument nor returns any value therefore it cannot be overloaded.
* Destructor cannot be declared as static and const;
* Destructors should be declared in the public section of the program.
* Destructor is called in the reverse order of its constructor invocation.

1. **What is RVO(Return Value Optimization) ?**

RVO is a compiler optimization that involves eliminating the temporary object created to hold a function return value.

1. **Initialization of data members in class?**

class variables are initialized in the same order as they appear in the class declaration.

**Example**: The program prints the correct value of x, but some garbage value for y, because y is initialized before x as it appears before in the class declaration.

#include<iostream>

using namespace std;

class Test {

private:

int y;

int x;

public:

Test() : x(10), y(x + 10) {}

void print();

};

void Test::print()

{

cout<<"x = "<<x<<" y = "<<y;

}

int main()

{

Test t;

t.print();

getchar();

return 0;

}

1. **What is constexpr?**

Constexpr is a feature added in c++11. The main idea is a performance improvement of the programs by doing computation at compile time rather than runtime.

constexpr specifies that the value of an object or a function can be evaluated at compile time and the expression can be used in other constant expressions.

**A function be declared as constexpr**

* In C++ 11, a constexpr function should contain only one return statement. C++ 14 allows more than one statement.
* constexpr function should refer only to constant global variables.
* constexpr function can call only other constexpr functions not simple functions.
* The function should not be of a void type and some operators like prefix increment (++v) are not allowed in the constexpr function.

**constexpr vs inline**

|  |  |
| --- | --- |
| **constexpr** | **inline** |
| It removes the function calls as it evaluates the code/expression in compile time | It hardly removes the function call as it acts on expression at runtime. |
| It is possible to assess the value of the variable or function at compile time. | It is not possible to access the value of the function or variable at compile time. |
| It doesn’t imply external linkage | It implies the external linkage |

**constexpr with constructors**: A constructor that is declared with a constexpr specifier is a constexpr constructor also constexpr can be used in the making of constructors and objects. A constexpr constructor is implicitly inline.

**Restrictions on constructors that can use constexpr:**

* No virtual base class
* Each parameter should be literal
* It is not trying to block the function

**A constexpr variable must satisfy the following requirements:**

* + It type must be literal type.
  + It must be immediately initialized
  + the [full](https://en.cppreference.com/w/cpp/language/eval_order) expression of its initialization, including all implicit conversions, constructors calls, etc, must be a [constant expression](https://en.cppreference.com/w/cpp/language/constant_expression).

**A constexpr function must satisfy the following requirements:**

* It must not be **virtual**.
* It must not be a **coroutine**.
* its return type (if any) must be a **LiteralType**
* each of its parameters must be a **LiteralType**
* for constructor and destructor (since C++20), the class must have no virtual base classes.
* there exists at least one set of argument values such that an invocation of the function could be an evaluated subexpression of a core constant expression (for constructors, use in a constant initializer is sufficient) (since C++14). No diagnostic is required for a violation of this bullet.
* its function body must not be a function-try-block
* the function body must be either deleted or defaulted or contain only the following:
* null statements (plain semicolons)
* static\_assert declarations
* typedef declarations and alias declarations that do not define classes or enumerations
* using declarations
* using directives
* if the function is not a constructor, exactly one return statement.
* the function body must not contain:
* a goto statement
* a statement with a label other than case and default
* a try-block
* an asm declaration
* a definition of a variable for which no initialization is performed
* a definition of a variable of non-literal type
* a definition of a variable of static or thread storage duration

1. what is consteval?

c++ 20 introduces the consteval, which is used to indicate that the function must be evaluated at compile time otherwise compiler error will be the result. Such functions are called immediate functions.

**Note**: use consteval, If you have a function that must run at compile time for some reason.

**Example**:

**#include <iostream>**

**using namespace std;**

**consteval int greater(int x, int y)**

**{**

**return (x>y? x:y);**

**}**

**int main()**

**{**

**constexpr g{greater(5, 10)};**

**cout<<g<<" is greater "<<endl;**

**return 0;**

**}**

**C++ 11 Features:**

**1) auto :** It is introduced in c++11, with auto we can declare a variable without specifying its type. Its type will be deduced the by data to which it’s initialized.

**Example:** Here type of **var\_1** will be **int** and the type of **var\_1** will be **char**.

// Storing an int inside an auto variable

auto var\_1 = 5;

// Storing a character inside a auto variable

auto var\_2 = 'C';

std::cout<<var\_1<<std::endl;

std::cout<<var\_2<<std::endl;

We can store any type in auto even if it is a function or some iterator. Storing the lambda function inside an auto variable.

**Example**:



// Storing Lambda function inside a auto variable

auto fun\_sum = [](int a , int b){

return a+b;

};

std::cout<<fun\_sum(4,5)<<std::endl;

The main advantage of auto comes with types that are very long to write i.e. Suppose we have std::map of std::string

Important points about auto variable in C++11

* Once you have initialized the auto variable then you can change the value, but you cannot change the type.
* It cannot be left uninitialized.

### Returning an auto from a function

To return an auto variable from a function we need to declare it in a special way in c++ 11

**auto sum(int x, int y) -> int**

**{**

**return x + y;**

**}**

But in c++ 14 this special way declaration is not needed we can declare the auto return function as a normal function.

Example :

**auto sum(int x, int y)**

**{**

**return x + y;**

**}**

**What is decltype?**

It inspects the declared type of an entity or the type of an expression

**What is typeid ?**

Typeid is an operator which is used where the dynamic type of an object needs to be known.

**Decltype vs typeid**

* Decltype gives the type information at compile time while typeid gives it at runtime.
* So, if we have a base class reference (or pointer) referring to (or pointing to) a derived class object, the decltype would give type as base class reference (or pointer, but typeid would give the derived type reference (or pointer).

**What is the default keyword?**

If you do not want to define the default constructor explicitly, we can define the function as ‘=default’.

Using =default only with functions that can be generated by the compiler is meaningful.

So, it can only be used with constructors, copy constructors, and destructors.

**Example**: In the following example copy the constructor and default constructor defined using default keyword.

**#include <iostream>**

**using namespace std;**

**class sample**

**{**

**int x;**

**public:**

**sample(const sample &obj) = default;**

**sample() = default;**

**sample(int x):x(x)**

**{**

**cout<<"Inside the parameterized constructor : "<<x<<endl;**

**}**

**void print()**

**{**

**cout<<x<<endl;**

**}**

**};**

**int main()**

**{**

**sample obj(100);**

**sample obj2 = obj;**

**obj2.print();**

**return 0;**

**}**

**What is delete keyword ?**

**It is introduced in c++11, we can make functions uncallable by using the delete keyword.**

**Its practical use cases are,**

* **Deleting compiler-generated functions like copy constructor, assignment operators, move constructor, move assignment operator, and default constructor.**
* **Deleting member functions to prevent data loss conversions**
* **Restrict Object creation on Heap by deleting new operators for class**
* **Delete specific template specializations.**

**Example :**

**#include <iostream>**

**#include <string>**

**class User**

**{**

**int id;**

**std::string name;**

**public:**

**User(int userId, std::string userName) : id(userId), name(userName)**

**{}**

**// Copy Constructor is deleted**

**User(const User & obj) = delete;**

**// Assignment operator is deleted**

**User & operator = (const User & obj) = delete;**

**void display()**

**{**

**std::cout<<id << " ::: "<<name<<std::endl;**

**}**

**// Deleting a constructor that accepts a double as ID to prevent narrowing conversion**

**User(double userId, std::string userName) = delete ;**

**// Deleting a constructor that accepts a double as ID to prevent invalid type conversion**

**User(char userId, std::string userName) = delete ;**

**// Delete the new function to prevent object creation on heap**

**void \* operator new (size\_t) = delete;**

**};**

**int main()**

**{**

**User userObj(3, "John");**

**// Can not copy User object as copy constructor is deleted**

**//User obj = userObj;**

**/\***

**\* Creating User objects with double or char as ID will cause compile time error**

**\***

**User obj4(5.5, "Riti");**

**obj4.display();**

**User obj5('a', "Riti");**

**obj5.display();**

**\*/**

**// Can not create object on heap as new operater is deleted**

**//User \* ptr = new User(1, "Riti");**

**return 0;**

**}**

## **Different between deleted functions and private functions**

By making functions private we can also restrict their calling but still marking function deleted with delete keyword has its own advantages like,

* Private member functions can be called from other member functions, whereas, deleted functions cannot be called even from other member functions.
* Deleted functions exists in name lookup , if compiler finds a function is deleted then it will not look up for other matching functions based on type lookups, hence prevents unnecessary data loss and bugs.

**What is final keyword** ?

Sometimes, you don’t want to allow the derived class to override the base class virtual function. In c++, allows the built-in facility to prevent overriding of virtual function using final specifier.

in C++ 11 can also be used to prevent inheritance of class / struct. If a class or struct is marked as final then it becomes non inheritable and it cannot be used as base class/struct.

**Example** :

**#include <iostream>**

**using namespace std;**

**class A final**

**{**

**public:**

**virtual void print() final**

**{**

**cout<<"printing in class A :"<<endl;**

**}**

**};**

**// class A is defined with final keyword then we cannot inherit the properties of class A**

**class B:public A**

**{**

**public:**

**void print()**

**{**

**cout<<"printing in class B : "<<endl;**

**}**

**};**

**int main()**

**{**

**B obj;**

**A \*obj1 = &obj;**

**obj1->print();**

**return 0;**

**}**

**What is the override keyword?**

In a member function declaration or definition, the override specifier ensures that the function is virtual and is overriding a virtual function from a base class. The program is ill-formed (a compile-time error is generated) if this is not true.

override is an identifier with special meaning when used after member function declarators: it's not a reserved keyword otherwise.

**What is the l-value reference ?**

lvalue is anything whose address is accessible. It means we can take the address of lvalue using & operator.

Example :

Int x = 10;

Int \*ptr = &x;

Int \*ptr1 = &(x+10); // compile error

## **What is rvalue ?**

Rvalue is anything that is not lvalue. It means we cannot take the address of **rvalue** and it also doesn’t persist beyond the single expression.

**What is Enum class in c++11?**

**Enum is a user-defined datatype that can be assigned some limited values. These values are defined by the programmer at the time of declaring the enumerated type.**

**Need for Enum Class over Enum Type:**

* **Two enumerations cannot share the same names.**
* **No variable can have a name that is already in some enumeration.**
* **Enums are not type-safe**

**Example:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

// Defining enum1 Gender

enum Gender { Male, Female };

**// Defining enum2 Gender2 with the same values This will throw an error**

enum Gender2 { Male,Female };

// Creating Gender type variable

Gender gender = Male;

**// creating a variable Male this will throw an error**

int Male = 100;

Gender2 gender2 = Female;

cout << gender << endl << gender2;

Gender gender = Male;

Color color = Red;

// Upon comparing gender and color it will return true as both have a value 0

// which should not be the case actually

if (gender == color)

cout << "Equal";

return 0;

}

### **Enum Class**: C++11 has introduced enum classes (also called **scoped enumerations**), which makes enumerations both **strongly typed and strongly scoped**. Class enum doesn’t allow implicit conversion to int, and also doesn’t compare enumerators from different enumerations. To define the enum we use the class keyword after the enum keyword.

**// Declaration**

**enum class EnumName{ Value1, Value2, ... ValueN};**

**// Initialisation**

**EnumName ObjectName = EnumName::Value;**

**Example** :

#include <bits/stdc++.h>

using namespace std;

int main()

{

// Defining enum1 Gender

enum class Gender { Male,

Female };

// Defining enum2 Gender2 with same values

// This will throw error

enum class Gender2 { Male,

Female };

// Creating Gender type variable

Gender gender = Gender::Male;

// creating a variable Male this will throw error

int Male = 100;

Gender2 gender2 = Gender2::Female;

cout << int(gender) << endl

<< int(gender2);

return 0;

}

**Delegation**:

Sometimes it is useful for a constructor to be able to call another constructor of the same class, This feature, called constructor delegation was introduced in c++11

**Multithreading**

**1) Thread creation in c++11?**

**i) Function pointer**

**ii) Function object**

**iii) Lambda functions.**

**Example:**

**#include <iostream>**

**#include <thread>**

**using namespace std;**

**class Threadclass**

**{**

**public:**

**void operator()()**

**{**

**for(int i = 0 ; i < 1000 ; i++)**

**cout<<"Thread object : "<<i<<endl;**

**}**

**};**

**void thread\_function()**

**{**

**for(int i = 0 ; i < 1000 ; i++)**

**cout<<"Thread\_Function : "<<i<<endl;**

**}**

**int main()**

**{**

**std::thread th(thread\_function); // thread creating using function pointer**

**std::thread th1((Threadclass())); // thread creating using function object**

**std::thread th2([](){ // thread creating using Lambda expression.**

**for(int i = 0 ; i < 1000; i++)**

**{**

**cout<<"Lambda function : "<<i<<endl;**

**}**

**});**

**th.join();**

**th1.join();**

**th2.join();**

**return 0;**

**}**

**2) what is join and detach ?**

**Join :** Once a thread is started then another thread can wait for this new thread to finish. For this another need to call join() function on the std::thread object

**Detach :** Detached threads are also called deamon/background threads. To detach a thread we need to call std::detach on std::thread.

#### **Be careful with calling detach() and join() on Thread Handles**

**Case 1: *Never call join() or detach() on std::thread object with no associated executing thread***

*Example :*

*std::thread threadObj( (WorkerThread()) );*

*threadObj.join();*

*threadObj.join(); // It will cause Program to Terminate*

*When a join() function is called on an thread object, then when this join(0 returns then that std::thread object has no associated thread with it. In case again join() function is called on such object then it will cause the program to Terminate.*

*Similarly calling detach() makes the std::thread object not linked with any thread function. In that case calling detach(0 function twice on an std::thread object will cause the program to terminate.*

*std::thread threadObj( (WorkerThread()) );*

*threadObj.detach();*

*threadObj.detach(); // It will cause Program to Terminate*

**Example :**

std::thread threadObj( (WorkerThread()) );

if(threadObj.joinable())

{

std::cout<<"Detaching Thread "<<std::endl;

threadObj.detach();

}

if(threadObj.joinable())

{

std::cout<<"Detaching Thread "<<std::endl;

threadObj.detach();

}

std::thread threadObj2( (WorkerThread()) );

if(threadObj2.joinable())

{

std::cout<<"Joining Thread "<<std::endl;

threadObj2.join();

}

if(threadObj2.joinable())

{

std::cout<<"Joining Thread "<<std::endl;

threadObj2.join();

}

# 3) Carefully Pass Arguments to Threads?

# To pass arguments to a thread’s associated callable object or function just pass additional arguments to std::thread constructor.

## **Passing simple arguments to a std::thread in C++11**

Example :

#include <iostream>

#include <thread>

using namespace std;

void threadcallback(int x, std::string str)

{

std::cout<<"Passed Number : "<<x<<endl;

std::cout<<"Passed String : "<<str<<endl;

}

int main()

{

int x = 10;

std::string str = "Sample String";

std::thread th(threadcallback,x,str);

th.join();

return 0;

}

## **How not to pass arguments to threads in C++11?**

Don’t pass addresses of variables from local stack to thread’s callback function. Because it might be possible that local variable in Thread 1 goes out of scope but Thread 2 is still trying to access it through it’s address.In such a scenario accessing an invalid address can cause unexpected behavior.  
For example,

**#include <iostream>**

**#include <thread>**

**using namespace std;**

**void threadfunction(int \*p)**

**{**

**std::cout<<"inside thread function : "<<p<<endl;**

**std::chrono::milliseconds dura(1000);**

**std::this\_thread::sleep\_for(dura);**

**\*p = 19;**

**}**

**void startnewthread()**

**{**

**int i = 10;**

**std::thread th(threadfunction, &i);**

**th.join();**

**cout<<"startNewthread : "<<i<<endl;**

**}**

**int main()**

**{**

**startnewthread();**

**std::chrono::milliseconds dura(1000);**

**std::this\_thread::sleep\_for(dura);**

**return 0;**

**}**

## **How to pass references to std::thread in C++11**

**#include <iostream>**

**#include <thread>**

**void threadCallback(int const & x)**

**{**

**int & y = const\_cast<int &>(x);**

**y++;**

**std::cout<<"Inside Thread x = "<<x<<std::endl;**

**}**

**int main()**

**{**

**int x = 9;**

**std::cout<<"In Main Thread : Before Thread Start x = "<<x<<std::endl;**

**std::thread threadObj(threadCallback, x);**

**threadObj.join();**

**std::cout<<"In Main Thread : After Thread Joins x = "<<x<<std::endl;**

**return 0;**

**}**

## **Assigning pointer to a member function of a class as thread function:**

Example :

**#include <iostream>**

**#include <thread>**

**class DummyClass {**

**public:**

**DummyClass()**

**{}**

**DummyClass(const DummyClass & obj)**

**{}**

**void sampleMemberFunction(int x)**

**{**

**std::cout<<"Inside sampleMemberFunction "<<x<<std::endl;**

**}**

**};**

**int main() {**

**DummyClass dummyObj;**

**int x = 10;**

**std::thread threadObj(&DummyClass::sampleMemberFunction,&dummyObj, x);**

**threadObj.join();**

**return 0;**

**}**

## **What is a Race Condition?**

# Race condition is one kind of bug that occurs in the multithread application.

# When two or more threads perform a set of operations in parallel, that access the same memory location.  Also, one or more thread out of them modifies the data in that memory location, then this can lead to an unexpected results some times.

## **Using mutex to fix Race Conditions?**

To fix race conditions in multi-threaded environment we need mutex i.e. each thread needs to lock a mutex before modifying or reading the shared data and after modifying the data each thread should unlock the mutex.

## std::mutex

In the C++11 threading library, the mutexes are in the <mutex> header file. The class representing a mutex is the std::mutex class.

There are two important methods of mutex:  
1.) lock()  
2.) unlock()

Example :

#include <iostream>

#include <thread>

#include <mutex>

#include <vector>

using namespace std;

class wallet

{

int m\_money;

std::mutex mut;

public:

wallet():m\_money(0){}

int getmoney()

{

return m\_money;

}

void addmoney(int money)

{

mut.lock();

for(int i = 0 ; i < money ; i++)

{

m\_money++;

}

mut.unlock();

}

};

int testMultithreadedWallet()

{

wallet walletobj;

std::vector<std::thread> threads;

for(int i = 0 ; i < 5 ; i++)

threads.push\_back(std::thread(&wallet::addmoney, &walletobj, 1000));

for(int i = 0 ; i < threads.size() ; i++)

threads.at(i).join();

return walletobj.getmoney();

}

int main()

{

int val = 0 ;

for(int k = 0; k < 1000; k++)

{

if((val = testMultithreadedWallet()) != 5000)

{

std::cout << "Error at count = "<<k<<" Money in Wallet = "<<val << std::endl;

//break;

}

}

return 0;

}

## **std::lock\_guard** :

std::lock\_guard is a class template, which implements the RAII for mutex.  
It wraps the mutex inside it’s object and locks the attached mutex in its constructor. When it’s destructor is called it releases the mutex.

In above program change the mutex.lock line as below.

**Std::lock\_guard<mutex> lock(mut);**

# Need of Event Handling

# Example :

# #include <iostream>

# #include <thread>

# #include <mutex>

# #include <chrono>

# using namespace std;

# class Application

# {

# bool m\_bloaded;

# std::mutex mut;

# 

# public:

# Application():m\_bloaded(false){}

# 

# void loaddata()

# {

# 

# std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

# std::cout<<"Loading data from xml : "<<endl;

# 

# std::lock\_guard<std::mutex> lock\_guard(mut);

# m\_bloaded = true;

# 

# }

# 

# void mainTask()

# {

# std::cout<<"doing some hand shake : "<<endl;

# 

# mut.lock();

# 

# while(m\_bloaded != true)

# {

# 

# mut.unlock();

# 

# std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

# 

# mut.lock();

# 

# }

# mut.unlock();

# 

# cout<<"Do processing on Load Data : "<<endl;

# 

# }

# };

# int main()

# {

# Application obj;

# std::thread th1(&Application::loaddata, &obj);

# std::thread th2(&Application::mainTask, &obj);

# 

# th1.join();

# th2.join();

# 

# return 0;

# }

**Condition Variable :**

Condition Variable is a kind of Event used for signaling between two or more threads. One or more thread can wait on it to get signaled, while another thread can signal this.

**#include <condition\_variable>**

A mutex is required along with condition variable.

**How things actually work with condition variable,**

* Thread 1 calls the wait on condition variable, which internally acquires the mutex and check if required condition is met or not.
* If not then it releases the lock and waits for Condition Variable to get signaled ( thread gets blocked). Condition Variable’s wait() function provides both these operations in atomic manner.
* Another Thread i.e. like Thread 2 signals the Condition Variable when condition is met
* Once Conditional Variable get signaled the the Thread 1 which was waiting for it resumes. It then acquires the mutex lock again and checks if the condition associated with Condition Variable is actually met or if it is superiors call. If more than one thread was waiting then notify\_one will unblock only one thread.
* If it was a superiors call then it again calls the wait() function.

**notify\_one()**

If any threads are waiting on same conditional variable object then  notify\_one unblocks one of the waiting threads.

**notify\_all()**

If any threads are waiting on same conditional variable object then  notify\_all unblocks all of the waiting threads.

Example:

**#include <iostream>**

**#include <thread>**

**#include <mutex>**

**#include <chrono>**

**#include <condition\_variable>**

**#include <functional>**

**using namespace std;**

**class Application**

**{**

**bool m\_bloaded;**

**std::mutex mut;**

**std::condition\_variable m\_condvar;**

**public:**

**Application():m\_bloaded(false){}**

**bool isDataloaded()**

**{**

**return m\_bloaded;**

**}**

**void loaddata()**

**{**

**std::this\_thread::sleep\_for(std::chrono::milliseconds(100));**

**std::cout<<"Loading data from xml : "<<endl;**

**std::lock\_guard<std::mutex> lock\_guard(mut);**

**m\_bloaded = true;**

**m\_condvar.notify\_one();**

**}**

**void mainTask()**

**{**

**std::cout<<"doing some hand shake : "<<endl;**

**std::unique\_lock<std::mutex> mlock(mut);**

**m\_condvar.wait(mlock);**

**cout<<"Do processing on Load Data : "<<endl;**

**}**

**};**

**int main()**

**{**

**Application obj;**

**std::thread th1(&Application::loaddata, &obj);**

**std::thread th2(&Application::mainTask, &obj);**

**th1.join();**

**th2.join();**

**return 0;**

**}**

# std::future , std::promise and Returning values from Thread :

**A std::future object can be used with asych, std::packaged\_task and std::promise. In this article will mainly focus on using std::future with std::promise object.**

**Many times we encounter a situation where we want a thread to return a result.**

**1.) Old Way : Share data among threads using pointer**

**Pass a pointer to the new thread and this thread will set the data in it. Till then in main thread keep on waiting using a condition variable. When new thread sets the data and signals the condition variable, then main thread will wake up and fetch the data from that pointer.**

**To do a simple thing we used a condition variable, a mutex and a pointer i.e. 3 items to catch a returned value.  
Now suppose we want this thread to return 3 different values at different point of time then problem will become more complex. Could there be a simple solution for returning the value from threads.**

**The answer is yes using std::future, checkout next solution for it.**

**2.) C++11 Way : Using std::future and std::promise**

**std::future is a class template and its object stores the future value.  
Now what the hell is this future value.**

**[showads ad=inside\_post]**

**Actually a std::future object internally stores a value that will be assigned in future and it also provides a mechanism to access that value i.e. using get() member function. But if somebody tries to access this associated value of future through get() function before it is available, then get() function will block till value is not available.**

**std::promise is also a class template and its object promises to set the value in future. Each std::promise object has an associated std::future object that will give the value once set by the std::promise object.**

**A std::promise object shares data with its associated std::future object.**

**Lets see step by step,**

**Create a std::promise object in Thread1.**

# std::promise<int> promiseObj;

**s of now this promise object doesn’t have any associated value. But it gives a promise that somebody will surely set the value in it and  
once its set then you can get that value through associated std::future object.**

**But now suppose Thread 1 created this promise object and passed it to Thread 2 object. Now how Thread 1 can know that when Thread 2 is going to set the value in this promise object?**

**The answer is using std::future object.**

**Every std::promise object has an associated std::future object, through which others can fetch the value set by promise.**

**So, Thread 1 will create the std::promise object and then fetch the std::future object from it before passing the std””promise object to thread 2 i.e.**

**std::future<**int**> futureObj = promiseObj.get\_future();**

**Example :**

**#include <iostream>**

**#include <thread>**

**#include <future>**

**using namespace std;**

**void initializer(std::promise<int> \*promObj)**

**{**

**cout<<"initializer : "<<endl;**

**promObj->set\_value(100);**

**}**

**int main()**

**{**

**std::promise<int> pobj;**

**std::future<int> futureObj = pobj.get\_future();**

**std::thread th(initializer, &pobj);**

**th.join();**

**cout<<"value : "<<futureObj.get()<<endl;**

**return 0;**

**}**

**STL (Standard Template Library) :**

**1) Map:** The Map is Associative container that stores the elements in a mapped fashion. Each element has a key value and mapped value. No two mapped values can have the same key value.

**#include <iostream>**

**#include <map>**

**using namespace std;**

**int main()**

**{**

**std::map<std::string,int> m\_map;**

**std::string strname = "vijaykumar";**

**if(m\_map.empty())**

**m\_map.insert(std::pair<std::string,int>("Rajkumar",10));**

**m\_map.emplace("Rajkumar",100);**

**m\_map[strname] = 20;**

**if(m\_map.find(strname) != m\_map.end())**

**m\_map[strname]++;**

**//printing the map values**

**for(auto it:m\_map)**

**{**

**cout<<it.first << " "<<it.second<<endl;**

**}**

**for(auto it:m\_map)**

**{**

**if(it.second <= 10)**

**m\_map.erase(it.first);**

**}**

**//assigning map value to another map**

**std::map<std::string,int> m\_map1(m\_map);**

**for(auto it:m\_map1)**

**cout<<it.first<<" "<<it.second<<endl;**

**return 0;**

**}**

**2) multimap:** multimap is similar to a map with the addition that multiple elements can have the same keys.

**Example :**

**#include <iostream>**

**#include <map>**

**#include <iterator>**

**#include <algorithm>**

**int main()**

**{**

**//MultiMap of char and int**

**std::multimap<std::string, int> mmapOfPos ={**

**{"Rajkumar", 1},**

**{"vijaykumar", 2},**

**{"Rajkumar", 3},**

**{"vijaykumar", 4},**

**{"Rajkumar", 5},**

**{"vijaykumar", 6},**

**{"Rajkumar", 7},**

**};**

**typedef std::multimap<std::string, int>::iterator MMAPIterator;**

**// It returns a pair representing the range of elements with key equal to 'c'**

**std::pair<MMAPIterator, MMAPIterator> result = mmapOfPos.equal\_range("Rajkumar");**

**std::cout << "All values for key 'c' are," << std::endl;**

**// Iterate over the range**

**for (MMAPIterator it = result.first; it != result.second; it++)**

**std::cout << it->second << std::endl;**

**// Total Elements in the range**

**int count = std::distance(result.first, result.second);**

**std::cout << "Total values for key 'c' are : " << count << std::endl;**

**return 0;**

**}**