1. What is a constructor

A constructor is a special type of member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object (instance of class) create. It is special member function of the class because it does not have any return type.

**How constructors are different from a normal member function?**

A constructor is different from normal functions in following ways:

* Constructor has same name as the class itself
* Constructors don’t have return type
* A constructor is automatically called when an object is created.
* It must be placed in public section of class.
* If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).

**Member fn::🡪**

**A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.**

**Member functions can be defined within the class definition or separately using scope resolution operator, : −. Defining a member function within the class definition declares the function inline, even if you do not use the inline specifier.**

[**Default Constructors:**](https://www.geeksforgeeks.org/c-internals-default-constructors-set-1/) Default constructor is the constructor which doesn’t take any argument. It has no parameters.

// Cpp program to illustrate the

// concept of Constructors

#include <iostream>

using namespace std;

class construct

{

public:

int a, b;

// Default Constructor

construct()

{

a = 10;

b = 20;

}

};

int main()

{

// Default constructor called automatically

// when the object is created

construct c;

cout << "a: " << c.a << endl

<< "b: " << c.b;

return 1;

}

Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

**2. 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.

// CPP program to illustrate

// parameterized constructors

#include <iostream>

using namespace std;

class Point

{

private:

int x, y;

public:

// Parameterized Constructor

Point(int x1, int y1)

{

x = x1;

y = y1;

}

int getX()

{

return x;

}

int getY()

{

return y;

}

};

int main()

{

// Constructor called

Point p1(10, 15);

// Access values assigned by constructor

cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

return 0;

}

**Uses of Parameterized constructor:**

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

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments.This concept is known as Constructor Overloading and is quite similar to [function overloading](https://www.geeksforgeeks.org/function-overloading-c/). 

* Overloaded constructors essentially have the same name (exact name of the class) and differ by number and type of arguments.
* A constructor is called depending upon the number and type of arguments passed.
* While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

// C++ program to illustrate

// Constructor overloading

#include <iostream>

using namespace std;

class construct

{

public:

    float area;

    // Constructor with no parameters

    construct()

    {

        area = 0;

    }

    // Constructor with two parameters

    construct(int a, int b)

    {

        area = a \* b;

    }

    void disp()

    {

        cout<< area<< endl;

    }

};

int main()

{

    // Constructor Overloading

    // with two different constructors

    // of class name

    construct o;

    construct o2( 10, 20);

    o.disp();

    o2.disp();

    return 1;

}

A copy constructor is a member function that initializes an object using another object of the same class. A copy constructor has the following general function prototype.

ClassName (const ClassName &old\_obj);

#include<iostream>

using namespace std;

class Point

{

private:

    int x, y;

public:

    Point(int x1, int y1)

{

x = x1; y = y1;

}

    // Copy constructor

    Point(const Point &p1)

{

x = p1.x; y = p1.y;

}

    int getX()            {  return x; }

    int getY()            {  return y; }

};

int main()

{

    Point p1(10, 15); // Normal constructor is called here

    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors

    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;

}

What happens when the constructor is private?

f a constructor is private, attempting to create an object of that class from outside the class using the usual syntax (e.g., `MyClass obj;`) will result in a compiler error. The only way to create an object of such a class would be to call a static member function or a friend function that is able to create the object by calling the private constructor.

Here is an example of a class with a private constructor and a static member function that creates objects of that class:

```

class MyClass {

private:

int x;

MyClass(int val) : x(val) {} // private constructor

public:

static MyClass create(int val) {

return MyClass(val); // can create objects by calling private constructor

}

Void disp()

{

Cout<<x;

}

};

```

In this example, the constructor of `MyClass` is declared as private, and the only way to create objects of `MyClass` is by calling the static member function `create()`, which can access the private constructor.

```

MyClass obj = MyClass::create(42); // create object using static member function

```

In this way, the private constructor allows for stricter control over object creation and can help enforce certain design patterns or access restrictions.

COPY CONSTRUCTOR VS COPY ASSIGNMENT OPERATOR

|  |
| --- |
| #include<iostream>  #include<stdio.h>  using namespace std;  class Test  {    public:      Test() {}      Test(const Test &t)      {       cout<<"Copy constructor called "<<endl; }        Test& operator = (const Test &t){          cout<<"Assignment operator called "<<endl;          return \*this }  };  // Driver code  int main()  {      Test t1, t2;      t2 = t1;      Test t3 = t1;      getchar();      return 0; }  } |

* **Output:**  
  *Assignment operator called  
  Copy constructor called*
* Copy constructor is called when a new object is created from an existing object, as a copy of the existing object (see [this](https://www.geeksforgeeks.org/g-fact-13/)G-Fact). And assignment operator is called when an already initialized object is assigned a new value from another existing object.

|  |
| --- |
| t2 = t1;  // calls assignment operator, same as "t2.operator=(t1);"  Test t3 = t1;  // calls copy constructor, same as "Test t3(t1);  **When is it Called:**   * The copy constructor is called in several situations, including:   + When an object is passed to a function by value.   + When an object is returned by value from a function.   + When an object is explicitly copied using the copy constructor.   void someFunction(MyClass obj); // Copy constructor called when obj is passed by value  MyClass createObject() {  MyClass newObj;  return newObj; // Copy constructor called when newObj is returned by value  }  MyClass originalObj;  MyClass copyObj(originalObj); // Copy constructor called when creating copyObj |

* the reason behind returning reference from copy assignment operator in c++ programming language. Is to support chaining assignment.
* Eg :

#include <iostream>

using namespace std;

class base{

private:

int a;

public:

base() {}

base(int val): a{val} {}

base& operator=(const base & t)

{

a=t.a; // important-------------------------------------

return \*this;

}

void print()

{

cout<< a<< endl;

}

};

int main(){

base b1(10);

base b2,b3,b4;

b2=b3=b4=b1;

b2.print();b3.print();b4.print();

return 0;

}

2nd example :: ------------------------------------------------------------------------

class MyClass {

public:

int a;

MyClass& operator=(const MyClass& other)

{

If(this!=& other)

{

a = other.a;

return \*this;

}

}

};

int main() {

MyClass obj1;

obj1.a = 10;

MyClass obj2;

obj2 = obj1; // copy assignment operator called

std::cout << obj2.a << std::endl; // output: 10

}

Why & is use din Assihment operator

1. **Chaining of Assignments:**
   * By returning a reference to the object (**\*this**), you allow for chaining of assignment operations. For example, **a = b = c** will work because the result of the assignment (**b = c**) is a reference to **b**, which is then assigned to **a**.

cppCopy code

MyClass a, b, c; a = b = c; // Chaining of assignments

1. **Consistency with Built-in Types:**
   * It makes the behavior consistent with built-in types like **int**, where assignments return a reference to the assigned variable.

cppCopy code

int x, y, z; x = y = z; // Chaining of assignments with built-in types

1. **Efficiency:**
   * Returning a reference allows the assignment operator to be used efficiently in expressions without unnecessary copying.

cppCopy code

MyClass a, b, c; a = b = c; // Efficient chaining of assignments

1. **Self-Assignment Check:**
   * When performing a self-assignment check (i.e., checking if the object is being assigned to itself), returning a reference allows you to avoid unnecessary work in the case of self-assignment.

cppCopy code

MyClass& operator=(const MyClass& other) { if (this != &other) { // Perform assignment of member variables data = other.data; } return \*this; // Return a reference to the current object }

In summary, returning a reference to the object being assigned is a common practice in C++ and provides consistency, enables chaining of assignments, and allows for more efficient and expressive use of the assignment operator. It is a key part of the "return by reference" idiom in C++.

**Can we make copy constructor private?**   
Yes, a copy constructor can be made private. When we make a 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. In such situations, we can either write our own copy constructor like above String example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

**Yes, you can make the copy constructor private in a C++ class. This is a technique often used to prevent objects of the class from being copied. When the copy constructor is declared as private, attempting to create a copy of an object will result in a compilation error.**

**#include <iostream>**

**class NoCopyClass {**

**private:**

**NoCopyClass(const NoCopyClass& other) {**

**std::cout << "Copy constructor is private. Cannot copy objects." << std::endl;**

**}**

**public:**

**// Public constructor**

**NoCopyClass() {**

**std::cout << "Default constructor called." << std::endl;**

**}**

**// Public member function**

**void someFunction() {**

**std::cout << "someFunction() called." << std::endl;**

**}**

**};**

**int main() {**

**NoCopyClass obj1;**

**// Uncommenting the line below will result in a compilation error**

**// NoCopyClass obj2 = obj1; // Error: 'NoCopyClass::NoCopyClass(const NoCopyClass&)' is private**

**obj1.someFunction(); // This is allowed**

**return 0;**

**}**

**It is useful in below situations:**

**Resource Management:**

* If your class manages unique resources (such as memory, file handles, network connections, etc.), making the copy constructor private ensures that copies of the object are not created accidentally. This helps prevent resource leaks and ensures proper ownership semantics.

**Immutable Objects:**

* In classes representing immutable objects (objects whose state cannot be modified after creation), making the copy constructor private ensures that instances remain immutable and cannot be changed through copying.

**Why argument to a copy constructor must be passed as a reference?**   
A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass an argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn’t allow parameters to be passed by value.

Passing the argument to a copy constructor as a reference is important for several reasons:

1. Avoiding unnecessary object creation: If the argument is passed by value (i.e., copied), it would invoke the copy constructor again, leading to an infinite loop. By passing the argument as a reference, you ensure that the copy constructor is only called once.

2. Efficiency: Passing by reference avoids the overhead of creating a new object and copying the entire contents of the original object. Instead, passing by reference allows the copy constructor to work directly with the original object's memory, avoiding unnecessary data duplication.

In summary, passing the argument to a copy constructor as a reference allows for efficient copying, avoids infinite loops, preserves const correctness, and ensures correct behavior with polymorphic objects.

**We cannot have virtual constructor**

**What is destructor?**   
Destructor is an instance member function which is invoked automatically whenever an object is going to destroy. Means, a destructor is the last function that is going to be called before an object is destroyed.

The thing to be noted is that destructor doesn’t destroys an object.

**Syntax:**

**~**constructor-name();

**Properties of Destructor:**

* Destructor function is automatically invoked when the objects are destroyed.
* It cannot be declared static or const.
* The destructor does not have arguments.
* It has no return type not even void.
* An object of a class with a Destructor cannot become a member of the union.
* A destructor should be declared in the public section of the class.
* The programmer cannot access the address of destructor.

**When is destructor called?**   
A destructor function is called automatically when the object goes out of scope:   
(1) the function ends   
(2) the program ends   
(3) a block containing local variables ends   
(4) a delete operator is called

**Can there be more than one destructor in a class?**   
No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

**When do we need to write a user-defined destructor?**   
If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

**Can a destructor be virtual?**   
Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function to void undefined behavior.

Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.

Hireachy of constructor

If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invokation is that the base class’s default constructor will be invoked first and then the derived class’s default constructor will be invoked.

#include <iostream>

using namespace std;

// base class

class Parent

{

    public:

    // base class constructor

    Parent()

    {

        cout << "Inside base class" << endl;

    }

};

// sub class

class Child : public Parent

{

    public:

    //sub class constructor

    Child()

    {

        cout << "Inside sub class" << endl;

    }

};

// main function

int main() {

    // creating object of sub class

    Child obj;

    return 0;

}

o/p= inside base class

inside sub class

**Order of constructor call for Multiple Inheritance**

For multiple inheritance order of constructor call is, the base class’s constructors are called in the order of inheritance and then the derived class’s constructor.

* C++

|  |
| --- |
| // C++ program to show the order of constructor calls  // in Multiple Inheritance    #include <iostream>  using namespace std;    // first base class  class Parent1  {        public:        // first base class's Constructor      Parent1()      {          cout << "Inside first base class" << endl;      }  };    // second base class  class Parent2  {      public:        // second base class's Constructor      Parent2()      {          cout << "Inside second base class" << endl;      }  };    // child class inherits Parent1 and Parent2  class Child : public Parent1, public Parent2  {      public:        // child class's Constructor      Child()      {          cout << "Inside child class" << endl;      }  };    // main function  int main() {        // creating object of class Child      Child obj1;      return 0;  } |

o/p=inside first base class’

inside seond base class

inside last base class

pgm to call parameetrised base constructor to that of parameterized constructor of derived class

#include <iostream>

using namespace std;

class base{

public:

base(int x)

{

cout<<"inside base class"<<x<<endl;

}

};

class der:public base{

public:

der(int z):base(50){

cout<<"inside derived class:"<<z<<endl;

}

};

int main(){

der d(8);

}

**How to call the parameterized constructor of base class in derived class constructor?**

To call the parameterized constructor of base class when derived class’s parameterized constructor is called, you have to explicitly specify the base class’s parameterized constructor in derived class as shown in below program:

* C++

|  |
| --- |
| // C++ program to show how to call parameterised Constructor  // of base class when derived class's Constructor is called    #include <iostream>  using namespace std;    // base class  class Parent {      int x;    public:      // base class's parameterised constructor      Parent(int i)      {          x = i;          cout << "Inside base class's parameterised "                  "constructor"               << endl;      }  };    // sub class  class Child : public Parent {      int j;    public:      // sub class's parameterised constructor      Child(int x)          : Parent(j)          , j(x)      {          cout << "Inside sub class's parameterised "                  "constructor"               << endl;      }  };    // main function  int main()  {        // creating object of class Child      Child obj1(10);      return 0;  } |

**Output:**

Inside base class's parameterised constructor

Inside sub class's parameterised constructor

2nd example:: --------------------------------------

#include <iostream>

using namespace std;

class Base {

protected:

int x;

public:

Base(int a) : x(a) {

cout << "Base class parameterized constructor called" << endl;

}

};

class Derived : public Base {

public:

Derived(int a, int b) : Base(a) {

cout << "Derived class parameterized constructor called" << endl;

cout << "Value of x in Base class: " << x << endl;

cout << "Value of y in Derived class: " << b << endl;

}

};

int main() {

Derived d(10, 20);

return 0;

}

**Important Points**:

* Whenever the derived class’s default constructor is called, the base class’s default constructor is called automatically.
* To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.
* The parameterized constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterized constructor of sub class.

[**Destructors**](https://www.geeksforgeeks.org/destructors-c/) in C++ are called in the opposite order of that of Constructors.

1. Shallow Copy:

A shallow copy involves creating a new object and copying the member variables from the source object to the destination object. However, if the member variables of the source object include pointers, only the memory addresses of the pointed data are copied, rather than creating new copies of the data itself. Both the source and destination objects will point to the same data. This can lead to problems if the data is modified through one object, as it will affect the other object as well.

Here's an example illustrating shallow copy:

```cpp

#include <iostream>

using namespace std;

class ShallowCopy {

public:

int\* data;

ShallowCopy(int val) {

data = new int(val);

}

~ShallowCopy() {

delete data;

}

// Shallow copy constructor

ShallowCopy(const ShallowCopy& other) {

data = other.data;

}

};

int main() {

ShallowCopy obj1(42);

ShallowCopy obj2 = obj1;

cout << \*obj1.data << endl; // Output: 42

cout << \*obj2.data << endl; // Output: 42

\*obj1.data = 100;

cout << \*obj1.data << endl; // Output: 100

cout << \*obj2.data << endl; // Output: 100 (both objects point to the same data)

return 0;

}

Shallow copy == copy consturtor;

In the above example, the `ShallowCopy` class has a member variable `data` that is dynamically allocated using `new`. The shallow copy constructor simply assigns the pointer `data` from the source object to the destination object, resulting in two objects pointing to the same dynamically allocated memory.

When we modify the value of `\*obj1.data`, it affects both `obj1` and `obj2` because they both point to the same memory. This demonstrates the issue with shallow copying when dealing with dynamically allocated data.

2. Deep Copy:

A deep copy involves creating a completely new object and copying the values of all member variables, including any dynamically allocated data. This ensures that the source and destination objects are completely independent and modifications to one object do not affect the other.

Here's an example illustrating deep copy:

```cpp

#include <iostream>

using namespace std;

class DeepCopy {

public:

int\* data;

DeepCopy(int val) {

data = new int;

\*data = val;

}

~DeepCopy() {

delete data;

}

// Deep copy constructor

DeepCopy(const DeepCopy& other) {

data = new int;

\*data = \*(other.data);

// data = new int(\*(other.data));

}

// Deep copy assignment operator

DeepCopy& operator=(const DeepCopy& other) {

if (this != &other) {

// Allocate new memory and copy the value

delete data;

data = new int(\*other.data);

}

return \*this;

}

};

int main() {

DeepCopy obj1(42);

DeepCopy obj2 = obj1;

cout << \*obj1.data << endl; // Output: 42

cout << \*obj2.data << endl; // Output: 42

\*obj1.data = 100;

cout << \*obj1.data << endl; // Output: 100

cout << \*obj2.data << endl; // Output: 42 (objects have independent copies of data)

return 0;

}

```

In this example, the `DeepCopy` class has a member variable `data` that is dynamically allocated. The deep copy constructor creates a new `int` object and assigns it the same value as the `data` from the source object.

When we modify the value of `\*obj1.data`, it

**// C++ program for the above approach**

**#include <iostream>**

**using namespace std;**

**// Box Class**

**class box {**

**private:**

**int length;**

**int breadth;**

**int height;**

**public:**

**// Function that sets the dimensions**

**void set\_dimensions(int length1, int breadth1,**

**int height1)**

**{**

**length = length1;**

**breadth = breadth1;**

**height = height1;**

**}**

**// Function to display the dimensions**

**// of the Box object**

**void show\_data()**

**{**

**cout << " Length = " << length**

**<< "\n Breadth = " << breadth**

**<< "\n Height = " << height**

**<< endl;**

**}**

**};**

**// Driver Code**

**int main()**

**{**

**// Object of class Box**

**box B1, B3;**

**// Set dimensions of Box B1**

**B1.set\_dimensions(14, 12, 16);**

**B1.show\_data();**

**// When copying the data of object**

**// at the time of initialization**

**// then copy is made through**

**// COPY CONSTRUCTOR**

**box B2 = B1;**

**B2.show\_data();**

**// When copying the data of object**

**// after initialization then the**

**// copy is done through DEFAULT**

**// ASSIGNMENT OPERATOR**

**B3 = B1;**

**B3.show\_data();**

**return 0;**

**}**

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.

* Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
* They are mainly used to achieve[Runtime polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/)
* Functions are declared with a **virtual**keyword in base class.
* The resolving of function call is done at Run-time.

**Rules for Virtual Functions**

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of ba
4. se class type to achieve run time polymorphism.
5. The prototype of virtual functions should be the same in the base as well as derived class.
6. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
7. A class may have [virtual destructor](https://www.geeksforgeeks.org/virtual-destructor/) but it cannot have a virtual constructor.

No, in C++, classes cannot have virtual constructors. Virtual functions, including constructors, are a fundamental feature of object-oriented programming that enable polymorphism, but constructors work a bit differently.

When an object is created, its constructor is called to initialize the object's data members. Constructors don't follow the same rules as other functions when it comes to being virtual because they are responsible for creating objects and setting up their memory layout. If you could make a constructor virtual, it would introduce complexities and ambiguities in how objects are constructed and managed in memory.

#include <iostream>

using namespace std;

class base {

public:

    virtual void print()

   {

        cout << "print base class" << endl;

    }

    void show()

    {

        cout << "show base class" << endl;

    }

};

class derived : public base {

public:

    void print()

    {

        cout << "print derived class" << endl;

    }

    void show()

    {

        cout << "show derived class" << endl;

    }

};

int main()

{

    base\* bptr;

    derived d;

    bptr = &d;

    // virtual function, binded at runtime

    bptr->print();

    // Non-virtual function, binded at compile time

    bptr->show();

}

**o/p** print derived class

show base class

-------------------------------------------------------------------------------------------------

1. **Static Functions/Methods:** A static function is a function that belongs to the class itself, rather than to instances of the class. It can be called without creating an instance of the class. Static functions are not associated with any particular instance data and can't be overridden by derived classes. They are shared among all instances of the class.

The reason a virtual function cannot be static is that the purpose of virtual functions is to enable polymorphism, which requires determining the method to be called at runtime based on the object's actual type. Static methods, on the other hand, are not tied to instances and are the same across all instances of the class, so there is no notion of dynamic dispatch that is needed for virtual functions.

Frnd function:::

In C++, a friend function is a function that is granted access to the private and protected members of a class. It is declared inside a class, but it is not a member of that class. Friend functions are useful when you want a non-member function to have access to the private or protected data of a class without making it a member of that class.

In special cases when a class’s private data needs to be accessed directly without using objects of that class, we need friend functions.

### **Special features of friend functions:**

* A friend function does not fall within the scope of the class for which it was declared as a friend. Hence, functionality is [not limited to one class](https://www.xspdf.com/resolution/58547943.html).
* The friend function can be a member of another class or a function that is [outside the scope of the class](https://www.scribd.com/doc/45584877/c-and-c).
* A friend function can be declared in the private or public part of a class [without changing its meaning](https://www.scribd.com/doc/45584877/c-and-c).
* Friend functions are not called using objects of the class because they are not within the class’s scope.
* Without the help of any object, the friend function can be [invoked like a normal member function](https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_functions_qa1.htm).
* Friend functions can use objects of the class as arguments.
* A friend function cannot explicitly access member names directly. Every member name has to use the object’s name and dot operator.. For example, Doctor.pay where pay is [the object name](https://www.xspdf.com/resolution/1749311.html).
* class className{
* // Other Declarations
* friend returnType functionName(arg list);
* };

class MyClass {

private:

int privateData;

public:

MyClass(int data) : privateData(data) {}

// Declaration of a friend function

friend void FriendFunction(const MyClass& obj);

};

// Definition of the friend function

void FriendFunction(const MyClass& obj) {

std::cout << "Friend function accessed private data: " << obj.privateData << std::endl;

}

int main() {

MyClass obj(42);

FriendFunction(obj);

return 0;

}

#### **Friend Class**

A friend class can have access to the data members and functions of another class in [which it is declared as a friend](https://www.xspdf.com/resolution/52369155.html). They are used in situations where we want a certain class to have access to another class’s private and protected members.

Classes declared as friends to any another class will have all the member functions become friend functions to the friend class. Friend functions are used to work as a link between the classes

#include <iostream>

using namespace std;

class Box {

double width;

public:

friend void printWidth( Box b );

void setWidth( double wid );

};

// Member function definition

void Box::setWidth( double wid ) {

width = wid; }

// Note: printWidth() is not a member function of any class.

void printWidth( Box b) {

/\* Because printWidth() is a friend of Box, it can

directly access any member of this class \*/

cout << "Width of box : " << b.width <<endl;

}

// Main function for the program

int main() {

Box box;

// set box width without member function

box.setWidth(10.0);

// Use friend function to print the wdith.

printWidth( box );

return 0;

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

Width of box : 10

2nd example:::

#include <iostream>

using namespace std;

// forward declaration

class ClassY;

class ClassX {

int digit1;

// friend class declaration

friend class ClassY;

public:

// constructor to initialize num1 to 10

ClassX() : digit1(10) {}

};

class ClassY {

int digit2;

public:

// constructor to initialize num2 to 5

ClassY() : digit2(5) {}

// member function to multiply num1

// from ClassX with num2 from ClassY

int multiply() {

ClassX m;

return m.digit1 \* digit2;

}

};

int main() {

ClassY n;

cout << "Multiplication: " << n.multiply();

return 0;

}

1. Anither example of friend function
2. #include <iostream>
4. **using** **namespace** std;
6. **class** A
7. {
8. **int** x =5;
9. **friend** **class** B;           // friend class.
10. };
11. **class** B
12. {
13. **public**:
14. **void** display(A &a)
15. {
16. cout<<"value of x is : "<<a.x;
17. }
18. };
19. **int** main()
20. {
21. A a;
22. B b;
23. b.display(a);
24. **return** 0;

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions.

Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Here are some key points about the **this** pointer:

* The **this** pointer is automatically created and available in every non-static member function of a class.
* It is a hidden pointer that is not explicitly declared, and its name is always **this**.

Inside a member function, the **this** pointer can be used to access the member variables and member functions of the current object.

Eg:#include <iostream>

using namespace std;

class Box {

public:

// Constructor definition

Box(double l = 2.0, double b = 2.0, double h = 2.0) {

cout <<"Constructor called." << endl;

length = l;

breadth = b;

height = h;

}

double Volume() {

return length \* breadth \* height;

}

int compare(Box b) {

return this->Volume() > b.Volume();

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main(void) {

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

if(Box1.compare(Box2)) {

cout << "Box2 is smaller than Box1" <<endl;

} else {

cout << "Box2 is equal to or larger than Box1" <<endl;

}

return 0;

}

---------------------------------------------------------------------------------------------------------

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. 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.

Let us try the following example to understand the concept of static data members −

[Live Demo](http://tpcg.io/3uMG7c)

#include <iostream>

using namespace std;

class Box {

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

public:

static int objectCount;

// Constructor definition

Box(double l = 2.0, double b = 2.0, double h = 2.0) {

cout <<"Constructor called." << endl;

length = l;

breadth = b;

height = h;

// Increase every time object is created

objectCount++;

}

double Volume() {

return length \* breadth \* height;

}

};

// Initialize static member of class Box

int Box::objectCount = 0;

int main(void) {

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

// Print total number of objects.

cout << "Total objects: " << Box::objectCount << endl;

return 0;

}

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

Constructor called.

Constructor called.

Total objects: 2

A derived class inherits all base class methods with the following exceptions −

* Constructors, destructors and copy constructors of the base class.
* Overloaded operators of the base class.
* The friend functions of the base class.

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

* It can be used **to pass current object as a parameter to another method.**
* It can be used **to refer current class instance variable.**
* It can be used **to declare indexers.**

Eg:🡪

#include <iostream>

**using** **namespace** std;

**class** Employee {

**public**:

**int** id; //data member (also instance variable)

       string name; //data member(also instance variable)

**float** salary;

       Employee(**int** id, string name, **float** salary)

        {

**this**->id = id;

**this**->name = name;

**this**->salary = salary;

        }

**void** display()

        {

            cout<<id<<"  "<<name<<"  "<<salary<<endl;

        }

};

**int** main(**void**) {

    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Emplye

   Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee

   e1.display();

   e2.display();

**return** 0;

* }

Structureee::

1. #include <iostream>
2. **using** **namespace** std;
3. **struct** Rectangle
4. {
5. **int** width, height;
7. };
8. **int** main(**void**) {
9. **struct** Rectangle rec;
10. rec.width=8;
11. rec.height=5;
12. cout<<"Area of Rectangle is: "<<(rec.width \* rec.height)<<endl;
13. **return** 0;

## **C++ Struct Example: Using Constructor and Method**

Let's see another example of struct where we are using the constructor to initialize data and method to calculate the area of rectangle.

#include <iostream>

**using** **namespace** std;

**struct** Rectangle    {

**int** width, height;

  Rectangle(**int** w, **int** h)

    {

        width = w;

        height = h;

    }

**void** areaOfRectangle() {

    cout<<"Area of Rectangle is: "<<(width\*height); }

 };

**int** main(**void**) {

**struct** Rectangle rec=Rectangle(4,6);

    rec.areaOfRectangle();

**return** 0;

}

47

Difference between Structs and Classes:

* **Structs are value type** whereas **Classes are reference type**.
* **Structs are stored on the stack** whereas **Classes are stored on the heap**.
* Value types hold their value in memory where they are declared, but reference type holds a reference to an object memory.
* **Structs can not have destructors**, but classes can have destructors.
* **Structs can not have explicit parameterless constructors** whereas classes can.
* Structs don't support inheritance, but classes do. Both support inheritance from an interface.

The structure is a user-defined [data type](https://www.geeksforgeeks.org/c-data-types/) that is available in [C++](https://www.geeksforgeeks.org/c-plus-plus/).

* Structures are used to combine different types of data types, just like an [array](https://www.geeksforgeeks.org/array-data-structure/) is used to combine the same type of data types.

**typedef** **struct** GeekForGeeks {

**int** G1;

**char** G2;

**float** G3;

}

[**typedef**](https://www.geeksforgeeks.org/typedef-versus-define-c/) is a keyword that is used to assign a new name to any existing data-type. Below is the C++ program illustrating use of struct using **typedef**:

 A union is a type of structure that can be used where the amount of memory used is a key factor.

* Similarly to the structure, the union can contain different types of data types.
* Each time a new variable is initialized from the union it overwrites the previous in C language but in C++ we also don’t need this keyword and uses that memory location.
* This is most useful when the type of data being passed through [functions](https://www.geeksforgeeks.org/functions-in-c/) is unknown, using a union which contains all possible data types can remedy this problem.
* It is declared by using the keyword “**union**“.

#include <iostream>

// Define a union named MyUnion

union MyUnion {

int intValue;

float floatValue;

char charValue;

};

int main() {

// Declare a variable of type MyUnion

MyUnion myVar;

// Assign a value to the float member

myVar.floatValue = 42.12;

// Assign a value to the int member

myVar.intValue = 12;

// Assign a value to the char member

myVar.charValue = 's';

// Access and print the value using the float member

std::cout << "Float Value: " << myVar.floatValue << std::endl;

// Access and print the value using the char member

std::cout << "Char Value: " << myVar.charValue << std::endl;

// Access and print the value using the int member

std::cout << "Int Value: " << myVar.intValue << std::endl;

return 0;

}

Since all members share the same memory space, when you assign a value to **myVar.intValue**, it overwrites the memory that was previously used for **myVar.floatValue**. This leads to garbage values when you later try to access **myVar.floatValue**.

* Enum in C++ is a data type that contains fixed set of constants.
* It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The C++ enum constants are static and final implicitly.

## **Points to remember for C++ Enum**

* enum improves type safety
* enum can be easily used in switch
* enum can be traversed
* enum can have fields, constructors and methods
* enum may implement many interfaces but cannot extend any class because it internally extends Enum class

C++ Enums can be thought of as classes that have fixed set of constants.

#include <iostream>

**using** **namespace** std;

**enum** week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }

**int** main()

{

    week day;

     day = Friday;

    cout << "Day: " << day+1<<endl;

**return** 0;

}

Structure padding is a concept in C that adds the one or more empty bytes between the memory addresses to align the data in memory.

Eg:

1. **struct** student
2. {
3. **char** a;
4. **char** b;
5. **int** c;
6. } stud1;

How to avoid the structure padding in C?

The structural padding is an in-built process that is automatically done by the compiler. Sometimes it required to avoid the structure padding in C as it makes the size of the structure greater than the size of the structure members.

We can avoid the structure padding in C in two ways:

* **Using #pragma pack(1) directive**
* **Using attribute**
* #pragma pack(1)
* **struct** base
* {
* **int** a;
* **char** b;
* **double** c;
* };

1. **struct** base
2. {
3. **int** a;
4. **char** b;
5. **double** c;
6. }\_\_attribute\_\_((packed));  ;

## **C++ provides two types of references:**

* References to non-const values
* References as aliases

**Eg1:int** &ref=a;

**Eg2:**

1. **int** a=10;   // 'a' is a variable.
2. **int** &b=a; // 'b' reference to a.
3. **int** &c=a; // 'c' reference to a.

The processor does not read 1 byte at a time. It reads 1 word at a time.

**What does the 1 word mean?**

If we have a 32-bit processor, then the processor reads 4 bytes at a time, which means that 1 word is equal to 4 bytes.

In object-oriented programming, both concrete classes and abstract classes are used as building blocks for creating objects and defining the behavior and structure of those objects. However, there are some key differences between the two:

1. Definition: A concrete class is a class that can be instantiated, meaning you can create objects directly from it. It provides a complete implementation of all its methods and can be used as a blueprint for creating specific objects. On the other hand, an abstract class cannot be instantiated itself. It serves as a base class or blueprint for other classes and provides a common interface or set of methods that derived classes must implement.

2. Instantiation: Concrete classes can be directly instantiated using the "new" keyword, and you can create objects of that class with specific values and behavior. Abstract classes, being incomplete, cannot be instantiated directly. You can only create objects of concrete classes that inherit from the abstract class and provide implementations for the abstract methods.

3. Abstract Methods: Abstract classes can declare abstract methods, which are methods without any implementation. These methods serve as placeholders and must be implemented by any concrete class that inherits from the abstract class. Concrete classes, on the other hand, do not have abstract methods; they provide the complete implementation for all their methods.

4. Inheritance: Concrete classes can inherit from other classes, both concrete and abstract, and extend their functionality. They can also implement interfaces to define additional behavior. Abstract classes, as mentioned earlier, serve as base classes and are designed to be inherited by concrete classes. A concrete class can inherit from only one abstract class, but it can inherit from multiple interfaces.

5. Usage: Concrete classes are commonly used when you have a specific object or concept that can be directly instantiated and used. They provide a clear implementation of the object's behavior. Abstract classes are used when you want to define common behavior or characteristics that multiple related classes should have. They provide a contract or template for derived classes to follow.

To summarize, concrete classes are used to create specific objects with complete implementations, while abstract classes are used as base classes for other classes to inherit from and provide partial or common functionality.

**Some interesting facts about abstract class**

1) We can’t create an object of abstract class.

2)We can have pointers and references of abstract class type.

3)  If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

**Similarities between virtual function and pure virtual function**

1. These are the concepts of Run-time polymorphism.
2. Prototype i.e. Declaration of both the functions remains the same throughout the program.
3. These functions can’t be global or static.

Virtual pure virtual

|  |  |
| --- | --- |
| Definition is given in base class. | No definition is given in base class. |
| Base class having virtual function can be instantiated i.e. its object can be made. | Base class having pure virtual function becomes abstract i.e. it cannot be instantiated. |
| If derived class do not redefine virtual function of base class, then it does not affect compilation. | If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class. |
| All derived class may or may not redefine virtual function of base class. | All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class. |

**Need for Virtual Base Classes:** Consider the situation where we have one class **A** . This class **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.

# <https://www.geeksforgeeks.org/calling-virtual-methods-in-constructordestructor-in-cpp/?ref=rp>

Virtual destructor

Virtual Destructor in C++ is used to release or free the memory used by the child class (derived class) object when the child class object is being removed from the memory using the parent class's pointer object.

“A virtual destructor is a special type of destructor in C++ that is declared as virtual in the base class. It is used when you have a class hierarchy and you want to ensure that the destructors of derived classes are called properly when deleting objects through a base class pointer”.

Virtual destructors maintain the hierarchy of calling destructors from child class to parent class as the virtual keyword used in the destructor follows the concept of **late binding** or the run-time binding.

# In C++, all the floating point constants are treated as double not as a float.

1. #include<iostream>
2. **using** **namespace** std;
3. **void** fun(**int**);
4. **void** fun(**float**);
5. **void** fun(**int** i)
6. {
7. std::cout << "Value of i is : " <<i<< std::endl;
8. }
9. **void** fun(**float** j)
10. {
11. std::cout << "Value of j is : " <<j<< std::endl;
12. }
13. **int** main()
14. {
15. fun(12);
16. fun(1.2);
17. **return** 0;
18. }

Make class non-inheritable without using final keyword in C++! REQUIREMENT:

<https://www.geeksforgeeks.org/simulating-final-class-in-c/>

We need one class which will make our class as final class. Lets call that class Final class.

SOLUTION: a. Make default constructor of Final class as private.

b. Inherit Final class as virtual in our class which we want to make non-inheritable.

c. Make our class as friend inside Final class. (so that only our class can call the constructor of Final class, not the derived class)

indirect method of making class as non-inheritable

#include <iostream>

using namespace std;

class final{

private:

final(){}

friend class base;

};

class base : virtual public final{

public:

base(){}

};

class derived: public base{

public:

derived(){}

};

int main(){

derived d;

return 0;

}

In C++ 11 we can make the base class non-inheritable by using [**final**](https://www.geeksforgeeks.org/c-final-specifier/) specifier. For eg, the following code gives a compile error as the base class is declared as final.

#include <iostream>

using namespace std;

class base final{

public:

base(){}

};

class derived : public base{

public:

derived(){

cout<<"in der:"; }

};

int main(){

derived d;

return 0;}