**OOPS – S1**

* A Class is a user-defined data type that has data members and member functions.
* Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behavior of the objects in a Class.
* In the above example of class Car, the data member will be speed limit, mileage, etc and member functions can apply brakes, increase speed, etc.

**Class in C++** is a blueprint representing a group of objects which shares some common properties and behaviors.

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Objects take up space in memory and have an associated address like a record in pascal or structure or union. When a program is executed the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other’s data or code, it is sufficient to know the type of message accepted and the type of response returned by the objects.

**Encapsulation**

Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

Encapsulation also leads to *data abstraction or data hiding*. Using encapsulation also hides the data. In the above example, the data of any of the sections like sales, finance, or accounts are hidden from any other section.

 We cannot use the class as it is. We first have to create an object of the class to use its features. An **Object** is an instance of a Class.

**class ClassName {**

*access\_specifier:*

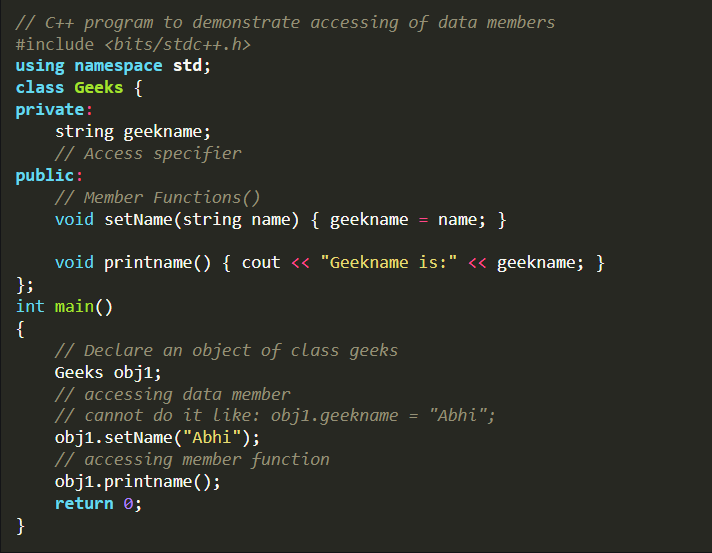
*// Body of the class*

**};**

ClassName ObjectName;

The data members and member functions of the class can be accessed using the dot(‘.’) operator with the object.

*obj.printName();*

**

If we do not specify the access specifier, the private specifier is applied to every member by default.

**There are 2 ways to define a member function:**

* Inside class definition
* Outside class definition

To define a member function outside the class definition,

* We have to first declare the function prototype in the class definition.
* Then we have to use the **scope resolution:: operator**along with the class name and function name.

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

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using the keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calls is reduced.

***Note:****Declaring a*[*friend function*](https://www.geeksforgeeks.org/friend-class-function-cpp/)*is a way to give private access to a non-member function.*

**Constructors**

[Constructors](https://www.geeksforgeeks.org/constructors-c/) are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition.

There are **4 types of constructors in C++ classes:**

* [Default Constructors](https://www.geeksforgeeks.org/constructors-c/): The constructor that takes no argument is called default constructor.
* Parameterized Constructors: This type of constructor takes the arguments to initialize the data members.
* [Copy Constructors](https://www.geeksforgeeks.org/copy-constructor-in-cpp/): Copy constructor creates the object from an already existing object by copying it.
* [Move Constructor:](https://www.geeksforgeeks.org/move-constructors-in-c-with-examples/) The move constructor also creates the object from an already existing object but by moving it.
* *// obj1 will call Default Constructor*
* Geeks obj1;
* cout <<"Geek id is: "<<obj1.id << endl;
* *// obj2 will call Parameterized Constructor*
* Geeks obj2(21);
* cout <<"Geek id is: " <<obj2.id << endl;

***Note:****If the programmer does not define the constructor, the compiler automatically creates the default, copy and move constructor.*

**Destructors**

[Destructor](https://www.geeksforgeeks.org/destructors-c/) is another special member function that is called by the compiler when the scope of the object ends.

It deallocates all the memory previously used by the object of the class so that there will be no memory leaks.

int main()

{

Geeks obj1;

obj1.id=7;

int i = 0;

**while** ( i < 5 )

{

Geeks obj2;

obj2.id=i;

i++;

} *// Scope for obj2 ends here*

**return** 0;

} *// Scope for obj1 ends here*

**Interesting Fact (Rare Known Concept)**

**Why do we give semicolons at the end of class?**

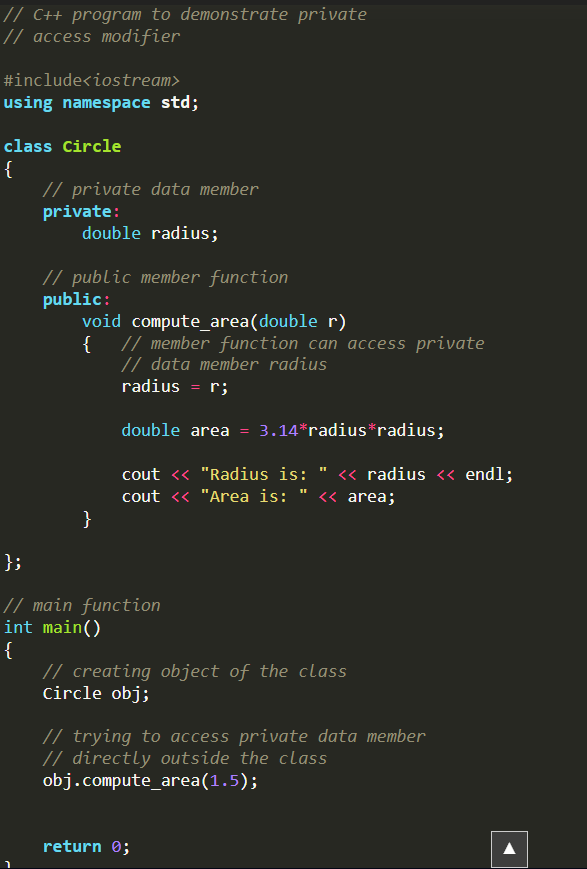
Many people might say that it’s a basic syntax and we should give a semicolon at the end of the class as its rule defines in cpp. But the main reason why semi-colons are there at the end of the class is compiler checks if the user is trying to create an instance of the class at the end of it.

Yes just like structure and union, we can also create the instance of a class at the end just before the semicolon. As a result, once execution reaches at that line, it creates a class and allocates memory to your instance.

<https://www.geeksforgeeks.org/c-classes-and-objects/?ref=lbp> for above semicolon part

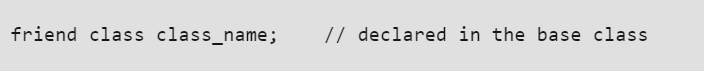
So by creating an instance just before the semicolon, we can create the Instance of class.

Access modifiers are used to implement an important aspect of Object-Oriented Programming known as **Data Hiding**.



**Friend Class and Function in C++**

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



A screen shot of a computer program

Description automatically generated

***A screenshot of a computer program

Description automatically generated***

***Note:****We can declare friend class or function anywhere in the base class body whether its private, protected or public block. It works all the same.*

// C++ Program to demonstrate the

// functioning of a friend class

#include <iostream>

using namespace std;

class GFG {

private:

int private\_variable;

protected:

int protected\_variable;

public:

GFG()

{

private\_variable = 10;

protected\_variable = 99;

}

// friend class declaration

friend class F;

};

// Here, class F is declared as a

// friend inside class GFG. Therefore,

// F is a friend of class GFG. Class F

// can access the private members of

// class GFG.

class F {

public:

void display(GFG& t)

{

cout << "The value of Private Variable = "

<< t.private\_variable << endl;

cout << "The value of Protected Variable = "

<< t.protected\_variable;

}

};

// Driver code

int main()

{

GFG g;

F fri;

fri.display(g);

return 0;

}

**Friend Function**

Like a friend class, a friend function can be granted special access to private and protected members of a class in C++. They are not the member functions of the class but can access and manipulate the private and protected members of that class for they are declared as friends.

A friend function can be:

1. **A global function**
2. **A member function of another class**

**Syntax:**

friend return\_type function\_name (arguments); // for a global function  
 or  
friend return\_type class\_name::function\_name (arguments); // for a member function of another class



***// C++ program to create a member function of another class***

***// as a friend function***

***#include <iostream>***

***using namespace std;***

***class base; // forward definition needed***

***// another class in which function is declared***

***class anotherClass {***

***public:***

***void memberFunction(base& obj);***

***};***

***// base class for which friend is declared***

***class base {***

***private:***

***int private\_variable;***

***protected:***

***int protected\_variable;***

***public:***

***base()***

***{***

***private\_variable = 10;***

***protected\_variable = 99;***

***}***

***// friend function declaration***

***friend void anotherClass::memberFunction(base&);***

***};***

***// friend function definition***

***void anotherClass::memberFunction(base& obj)***

***{***

***cout << "Private Variable: " << obj.private\_variable***

***<< endl;***

***cout << "Protected Variable: " << obj.protected\_variable;***

***}***

***// driver code***

***int main()***

***{***

***base object1;***

***anotherClass object2;***

***object2.memberFunction(object1);***

***return 0;***

***}***

***Note****: The order in which we define the friend function of another class is important and should be taken care of. We always have to define both the classes before the function definition. Thats why we have used out of class member function definition.*

If a non-friend function tried to create an object using the private constructor, it would result in a compilation error.

Friend functions are particularly useful when constructors are private or protected, as they allow controlled access to the creation of objects.

In essence, friend functions have special privileges and can interact with all members of the class, including constructors, regardless of their access specifier.

**Features of Friend Functions**

* A friend function is a special function in C++ that in spite of not being a member function of a class has the privilege to **access** the **private and protected data** of a class.
* A friend function is a non-member function or ordinary function of a class, which is declared as a friend using the keyword “**friend**” inside the class. By declaring a function as a friend, all the access permissions are given to the function.
* The keyword “friend” is placed only in the function declaration of the friend function and **not** in the **function definition or call.**
* A friend function is called like an ordinary function. It cannot be called using the object name and dot operator. However, it may accept the object as an argument whose value it wants to access.
* A friend function can be declared in any section of the class i.e. public or private or protected.
* **A Function Can Be Friendly to Multiple Classes**

**Advantages of Friend Functions**

* A friend function is able to access members without the need of inheriting the class.
* The friend function acts as a bridge between two classes by accessing their private data.
* It can be used to increase the versatility of overloaded operators.
* It can be declared either in the public or private or protected part of the class.

**Disadvantages of Friend Functions**

* Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
* Friend functions cannot do any run-time polymorphism in their members.

**Important Points About Friend Functions and Classes**

1. Friends should be used only for limited purposes. Too many functions or external classes are declared as friends of a class with protected or private data access lessens the value of encapsulation of separate classes in object-oriented programming.
2. Friendship is **not mutual**. If class A is a friend of B, then B doesn’t become a friend of A automatically.
3. Friendship is not inherited. (See [this](https://www.geeksforgeeks.org/g-fact-34/)for more details)
4. The concept of friends is not in Java.

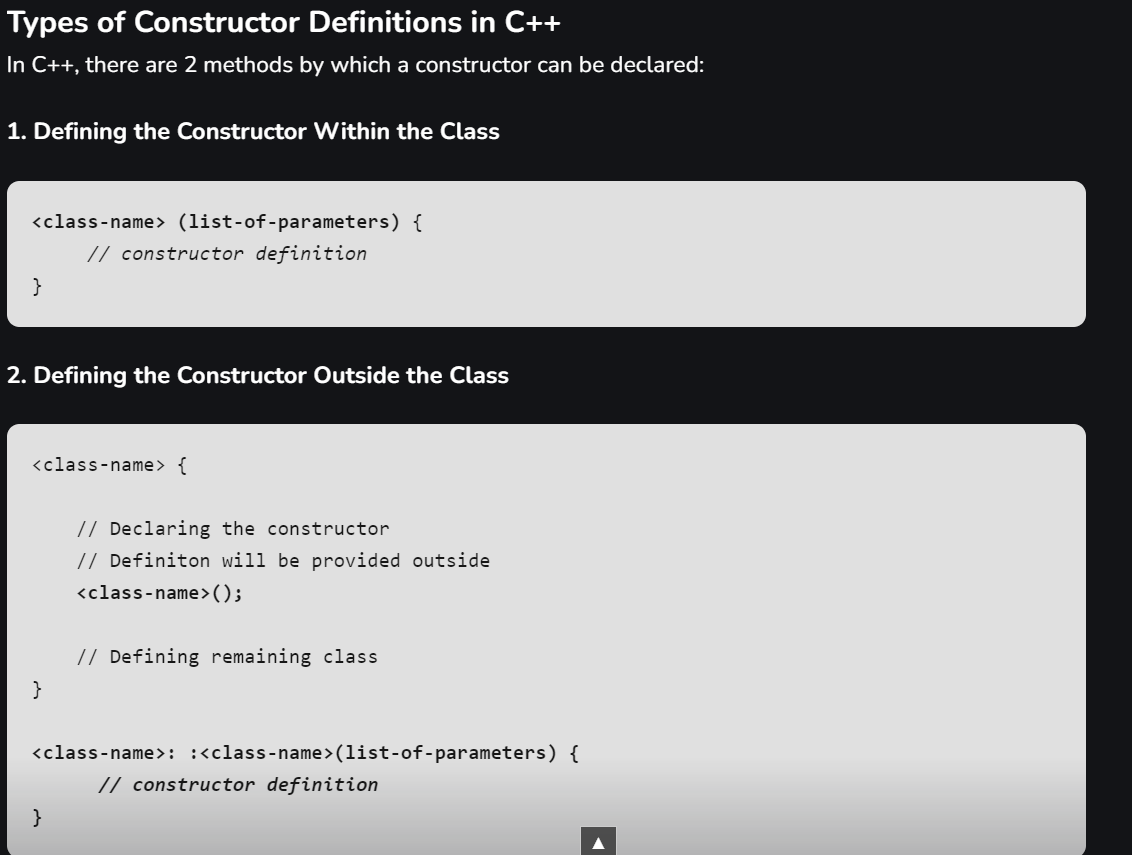
**Constructors in C++**

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

<class-name> (){  
...  
}

**Characteristics of Constructors in C++**

* 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.



^like member functions outside class are declared.

***Note:****We can make the constructor defined outside the class as****inline****to make it equivalent to the in class definition. But note that*[*inline*](https://www.geeksforgeeks.org/inline-functions-cpp/)*is not an instruction to the compiler, it is only the request which compiler may or may not implement depending on the circumstances.*

[Types of Constructors in C++](https://www.geeksforgeeks.org/types-of-constructors-in-cpp/)

Constructors can be classified based on in which situations they are being used. There are 4 types of constructors in C++:

1. **Default Constructor**: No parameters. They are used to create an object with default values. It is also called a zero-argument constructor.

If the default constructor is defined explicitly in the program by the programmer

1. **Parameterized Constructor**: Takes parameters. Used to create an object with specific initial values.

If we want to initialize the data members, we can also use the [initializer list](https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/) as shown:

MyClass::MyClass(int val) : memberVar(val) {};

1. **Copy Constructor**: Takes a reference to another object of the same class. Used to create a copy of an object.
2. Just like the default constructor, the C++ compiler also provides an implicit copy constructor if the explicit copy constructor definition is not present.
3. Here, it is to be noted that, unlike the default constructor where the presence of any type of explicit constructor results in the deletion of the implicit default constructor, the implicit copy constructor will always be created by the compiler if there is no explicit copy constructor or explicit move constructor is present.
4. **Move Constructor**: Takes an rvalue reference to another object. Transfers resources from a temporary object.

*Yes, we can have more than one constructor in a class. It is called*[*Constructor Overloading*](https://www.geeksforgeeks.org/constructor-overloading-c/)*.*

***‘this’ pointer in C++***

*To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.*

1. *Each object gets its own copy of the data member.*
2. *All-access the same function definition as present in the code segment.*

***C++ Static Data Members***

* *It is initialized before any object of this class is created, even before the main starts outside the class itself.*
* *It is visible can be controlled with the class access specifiers.*
* *Its lifetime is the entire program.*

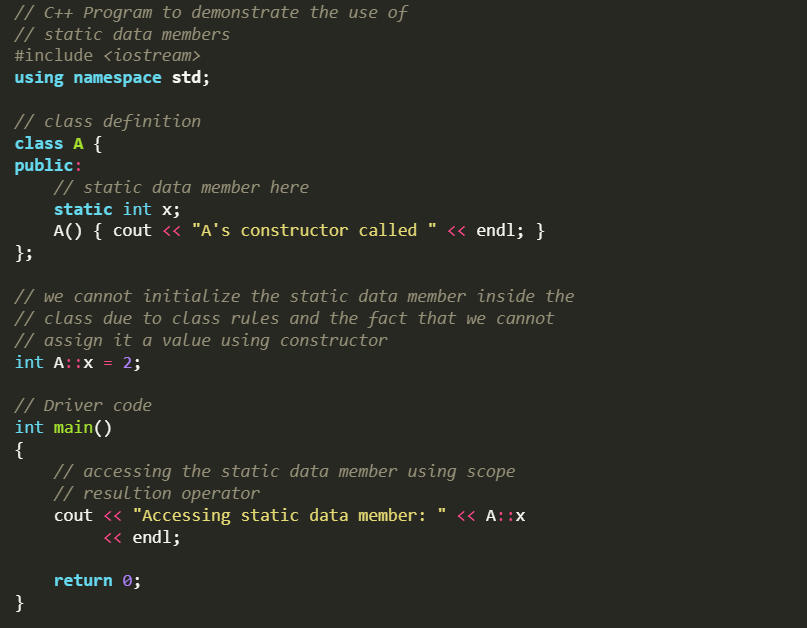
datatype class\_name::var\_name = value...;

***Note:****The static data members are initialized at compile time so the definition of static members should be present before the compilation of the program*

*Static data members must be defined outside the class definition.*

**Can we have static member functions in a class?**

*Yes,*[*static member functions*](https://www.geeksforgeeks.org/static-member-function-in-cpp/)*can be defined in a class. They can access static data members but cannot access non-static data members or this pointer.*



**Static Member in C++**

Static members of a class are not associated with the objects of the class. Just like a static variable once declared is allocated with memory that can’t be changed every object points to the same memory

Once a static member is declared it will be treated as same for all the objects associated with the class.

**Static Member Function in C++**

Static Member Function in a class is the function that is declared as static because of which function attains certain properties as defined below:

* A static member function is independent of any object of the class.
* A static member function can be called even if no objects of the class exist.
* A static member function can also be accessed using the class name through the scope resolution operator.
* A static member function can access static data members and static member functions inside or outside of the class.
* Static member functions have a scope inside the class and cannot access the current object pointer.
* You can also use a static member function to determine how many objects of the class have been created.

**The reason we need Static member function:**

* Static members are frequently used to store information that is shared by all objects in a class.
* For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter. This static data member can be increased each time an object is generated to keep track of the overall number of objects.

**‘this’ pointer in C++**

‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).

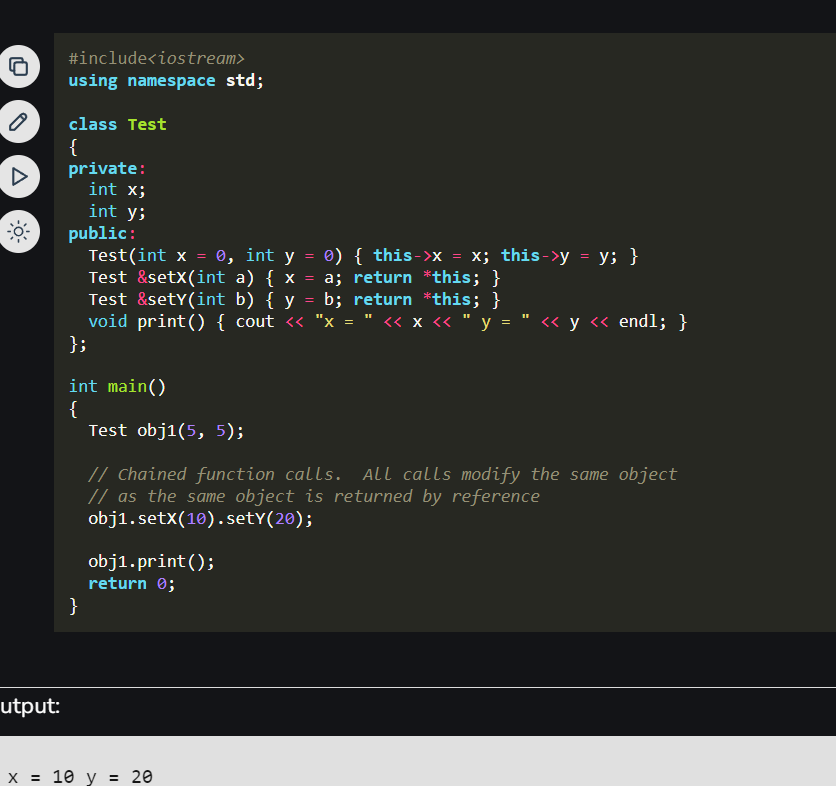
 Following are the situations where ‘this’ pointer is used:

**1) When local variable’s name is same as member’s name**

For constructors, [initializer list](https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/) can also be used when parameter name is same as member’s name.

**2) To return reference to the calling object**

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.



**Function Overloading in C++**

Function overloading can be considered as an example of a [polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/) feature in C++.

**How does Function Overloading work?**

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

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

               ->Float is promoted to double

* *If no match is found*:

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

* *ELSE ERROR*

***Note:****In C++, function overloading is primarily based on the function name and the types or number of its parameters. The return type alone does not distinguish overloaded functions. Therefore, changing only the return type of a function without changing its parameters does not create an overload and may lead to a compilation error due to ambiguity.*

**Functions that cannot be overloaded in C++**

In C++, following function declarations **cannot** be overloaded. 1) Function declarations that differ only in the return type. Ptogram fails in compilation.

2) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration. Program fails in compilation.

3) Parameter declarations that differ only in a pointer \* versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types. For example, following two function declarations are equivalent.

4) Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

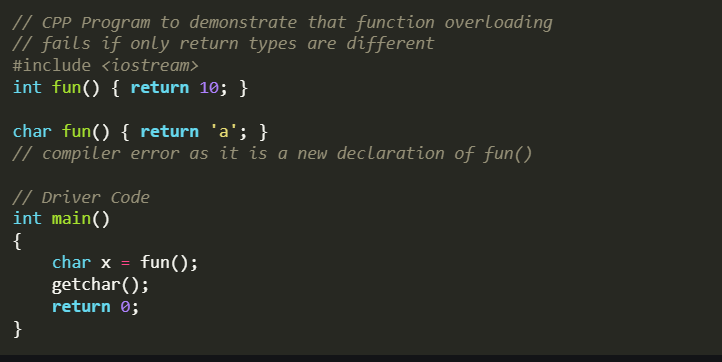
.

.

.

Function overloading comes under the compile-time polymorphism. During compilation, the function signature is checked. So, functions can be overloaded, if the signatures are not the same. The return type of a function has no effect on function overloading, therefore the same function signature with different return type will not be overloaded.

Note: However, function overloading with different return types and different parameter list is possible.(eg. if there are two functions: int sum(int a, int b) and float sum(float a, float b ) can be overloaded.



^ambiguity

**Inheritance in C++**

**class** *derived\_class\_name* : *access-specifier* *base\_class\_name*  
{  
 *// body ....*  
};

***Note****: A derived class doesn’t inherit****access****to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.*

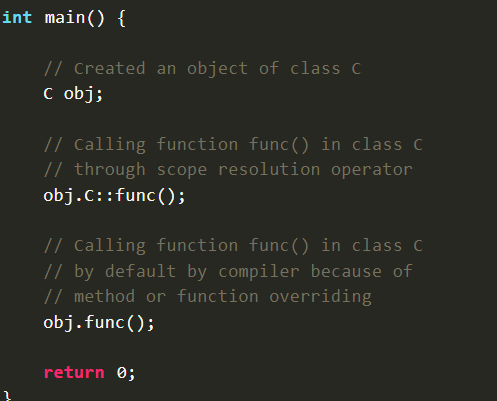
**Solution to  Ambiguity:**

To solve this ambiguity ***scope resolution operator*** is used denoted by ‘**::** ‘

**Syntax:**

ObjectName.ClassName::FunctionName();

**Function Overriding or Method Overriding**



**Syntax for Virtual Base Classes:**

**Syntax 1:**  
class B : virtual public A   
{  
};  
**Syntax 2:**  
class C : public virtual A  
{  
};

**Note:**   
**virtual** can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.