**Concept of Namespace:**

Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area if they live in different area or their mother or father name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

## Defining a Namespace:

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

namespace namespace\_name {

// code declarations

}

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

name::code; // code could be variable or function.

Let us see how namespace scope the entities including variable and functions:

#include <iostream>

using namespace std;

// first name space

namespace first\_space{

void func(){

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space{

void func(){

cout << "Inside second\_space" << endl;

}

}

int main ()

{

// Calls function from first name space.

first\_space::func();

// Calls function from second name space.

second\_space::func();

return 0;

}

If we compile and run above code, this would produce the following result:

Inside first\_space

Inside second\_space

## The using directive:

You can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code:

#include <iostream>

using namespace std;

// first name space

namespace first\_space{

void func(){

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space{

void func(){

cout << "Inside second\_space" << endl;

}

}

using namespace first\_space;

int main ()

{

// This calls function from first name space.

func();

return 0;

}

If we compile and run above code, this would produce the following result:

Inside first\_space

The using directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows:

using std::cout;

Subsequent code can refer to cout without prepending the namespace, but other items in the **std** namespace will still need to be explicit as follows:

#include <iostream>

using std::cout;

int main ()

{

cout << "std::endl is used with std!" << std::endl;

return 0;

}

If we compile and run above code, this would produce the following result:

std::endl is used with std!

Names introduced in a **using** directive obey normal scope rules. The name is visible from the point of the **using** directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

## Discontiguous Namespaces:

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files.

So, if one part of the namespace requires a name defined in another file, that name must still be declared. Writing a following namespace definition either defines a new namespace or adds new elements to an existing one:

namespace namespace\_name {

// code declarations

}

## Nested Namespaces:

Namespaces can be nested where you can define one namespace inside another name space as follows:

namespace namespace\_name1 {

// code declarations

namespace namespace\_name2 {

// code declarations

}

}

You can access members of nested namespace by using resultion operators as follows:

// to access members of namespace\_name2

using namespace namespace\_name1::namespace\_name2;

// to access members of namespace:name1

using namespace namespace\_name1;

In the above statements if you are using namespace\_name1, then it will make elements of namespace\_name2 available in the scope as follows:

#include <iostream>

using namespace std;

// first name space

namespace first\_space{

void func(){

cout << "Inside first\_space" << endl;

}

// second name space

namespace second\_space{

void func(){

cout << "Inside second\_space" << endl;

}

}

}

using namespace first\_space::second\_space;

int main ()

{

// This calls function from second name space.

func();

return 0;

}

If we compile and run above code, this would produce the following result:

Inside second\_space

**Stream I/O routines:**

These functions process data in different sizes and formats, from single characters to large data structures. They also provide buffering, which can improve performance. The default size of a stream buffer is 4K. These routines affect only buffers created by the run-time library routines, and have no effect on buffers created by the operating system.

### Some Stream I/O Routines:

|  |  |
| --- | --- |
| Routine | Use |
| |  | | --- | | [fclose](https://msdn.microsoft.com/en-us/library/fxfsw25t.aspx) | | Close stream |
| [feof](https://msdn.microsoft.com/en-us/library/xssktc6e.aspx) | Test for end of file on a stream |
| [fflush](https://msdn.microsoft.com/en-us/library/9yky46tz.aspx) | Flush stream to buffer or storage device |
| [fprintf, \_fprintf\_l, fwprintf](https://msdn.microsoft.com/en-us/library/xkh07fe2.aspx) | Write formatted data to stream |
| [printf, \_printf\_l, wprintf](https://msdn.microsoft.com/en-us/library/wc7014hz.aspx) | Write formatted data to stdout |
| [puts, \_putws](https://msdn.microsoft.com/en-us/library/tf52y4t1.aspx) | Write line to stream |
| [scanf, \_scanf\_l, wscanf](https://msdn.microsoft.com/en-us/library/9y6s16x1.aspx) | Read formatted data from stdin |
| [getc, getwc](https://msdn.microsoft.com/en-us/library/5231d02a.aspx) | Read character from stream (macro versions of **fgetc** and **fgetwc**) |

**Concept of OOP:**

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

There are a few **principle concepts** that form the foundation of object-oriented programming:

## Object

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

## Class

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

## Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

## Encapsulation

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

## Inheritance

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

## Polymorphism

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

## Overloading

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

**Object Oriented Programming Philosophy:**

### Introduction:

Speaking the terms of "philosophy" means the deep understanding of something. Therefore, OOP philosophy here means (for me) a deep understanding of OOP. So, I hope, after following this tutorial, you will be able to see clearly the gems behind OOP. Why it is preferred by so many people and why it is said to be a better solution in solving things.

### The Understanding

Hey, what does OOP mean? Yeah, we know its abbreviation, but what it is all about? It's a programming style that based its thinking into objects in the real world. That's how it got its name. In "normal" way in thinking programs, we usually think about the functionality of a program. It's like this: oh, this program calculate the degree of students. So, we usually divide the program based on its functionality, for example: We might build a procedure to get the students marks. Then, we build another function to determine calculate the final marks from that. Then we build a function to determine the grade of that particular final mark. And so on. This kind of habit severely infects our mind so that we cannot think into another perspective.

### Encapsulation

After you think which objects are involved in your program, you should list the following:

* What is the characteristics of that objects
* What values or properties convey that object
* What can the object do or what can be done to the object
* How is the interaction between objects
* How is the classification of the objects
* Of course you should list the ones that is relevant to your program. You can list that students can play football, but that's nothing to do with grade report program. :-) Students do some exams? Ummm yes... but it's not relevant either. Since students are not active in determining the grade, we should think the other way. Aha! Students can receive the report. OK, so we would like to code something like this:

Student.Receive(Report);

That's a pseudocode.

### Public or Private

Declaring things as public means that it is accessible by outsiders. The other hand, declaring things private means that it is invisible by outsiders, thus making it unable to be accessed by anyone but the instance internal. What is the "public" and "private" here means? They serve as access keywords. They determine the grant policy of the accesses into that particular thing.

### Constructing Method Body

Constructing method body is just like constructing procedures or functions.

### Constructors

Naturally, when we want to create an instance of an object, we would like to initialize some things (like assigning variables to some values, allocating some memory for buffers, etc) prior to utilizing the object. The OOP has this. We would have to make a **constructor** to do it. What is a constructor? Constructor is basically a method, specially invoked during the creation of the object instance. To make a constructor in Pascal is very simple just declare a method with a keyword **constructor**. That's it. Usually the method name is Init, but you can name it your own. Just keep in mind though, that Pascal's creation of constructor is somewhat unusual compared to other languages like C or C++.

### Destructors

**Destructors** behaves much similarly as constructors. Constructors are called during the creation of the object, but destructors are called during the destruction of the object. The purpose of having destructor is to destroy any memory allocation created by constructors. Suppose you have a constructor that allocates some memory for a buffer and store it in pointer p. After the object get destroyed, you have to destroy p as well. Otherwise, that portion of memory will floating around and occupies some space unused. The destructor then come in, invoked when the object got destroyed and clear such memory allocations.

### Inheritance

Classes can inherit the **behaviors** of other classes. The behavior I mentioned here means either methods or fields or both. Let's say there is a class A and class B. Class B inherits class A. Class B is therefore called the **sub-class** of class A. Similarly, class A is called the **super-class** of B.

What is the effect of inheritance anyway? Suppose class A has two methods -- foo and bar -- and three fields -- x, y, and z. If we declare class B to inherit class A, class B **will automatically have** foo, bar, x, y, and z altogether even though we do not declare them.

### Benefit of Inheritance

Then, what is the benefit of inheritance? This important property will allow you to model something from common types to its more specific types. For example, for your graphic library, you may want to create objects: Circle, Triangle, and Square. All of them will have the method: put -- to put the object into new coordinates, getX to get the X coordinate, similarly getY, setColor to set its color, and draw to draw itself.

### Method Overloading

One of the neat things of OOP features is called **method overloading**. That cool name is simply a term for **redefining ancestors methods** to suit the needs of the current class.  Suppose you have a class declaration Employee to model employees in a database system, then you also have a class ITStaff which is declared as a descendant of Employee.

### Polymorphism and Virtual Functions

What is polymorphism? As you may guess: It means "many forms". What is it exactly then? To answer this question, let's imagine that we are going to develop a graphic libraries that can draw rectangles, triangles, and circles using OOP technique. You quickly notice that those three are of one common parent object, let's call it Shape2D. For each shapes, we define various capabilities. Of course, we then put the common capabilities into the parent. Those capabilities are coded inside methods in parent object. Then, we inherit those common methods and override them for each shapes since rectangles behave differently from triangles and circles. Those common capabilities are further translated down into specific child objects. Let's say one of those is the method Draw.

Thus, when we pass on an object to a method that takes a Shape2D parameter, wouldn't it be nice if we can invoke the method Draw and the correct Draw method is invoked without being explicitly examine that parameter whether it is a rectangle, triangle, or circle.

### Abstract Classes

Class that contains virtual functions are called **abstract classes**. You **should not** instantiate abstract classes. In other words, don't new it to create an instance. Why? Because its virtual methods are not supposed to be called. Actually, the real OOP compiler should complain if an abstract classes is instantiated.

Abstract classes must have children classes. **All** of their virtual functions have to be inherited and overrided in the children classes. If not all of the virtual functions are inherited and overrided in a particular child class, then that child class will also be called an abstract class. If that's the case, that particular child class should be then be inherited by some other grandchildren classes.

### Closing

This is the last of four articles covering OOP philosophies. These are not exactly THE OOP philosophies, but I'd rather say the OOP features supported by Borland Pascal. However, the features covered here, fortunately, the key features of OOP. I should say that Java offer a more brilliant OOP features. If you're interested on OOP, you should leer on Java.

Building programs based on OOP techniques are very difficult at first. It may require "brainwashing" :-). Just kidding. Somehow, that's true since your old paradigm has to be overhauled to fit into a totally new idea. This can be daunting at first. However, I should say doing things at first are always difficult. So, I suggest you to keep trying and don't give up.

**Classes and Objects:**

## Class Definition:

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows:

class Box

{

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

## Objects Definition:

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members.

## Classes & Objects in Detail:

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below:

|  |  |
| --- | --- |
| **Concept** | **Description** |
| [Class member functions](http://www.tutorialspoint.com/cplusplus/cpp_class_member_functions.htm) | A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. |
| [Class access modifiers](http://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm) | A class member can be defined as public, private or protected. By default members would be assumed as private. |
| [Constructor & destructor](http://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm) | A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted. |
| [C++ copy constructor](http://www.tutorialspoint.com/cplusplus/cpp_copy_constructor.htm) | The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. |
| [C++ friend functions](http://www.tutorialspoint.com/cplusplus/cpp_friend_functions.htm) | A **friend** function is permitted full access to private and protected members of a class. |
| [C++ inline functions](http://www.tutorialspoint.com/cplusplus/cpp_inline_functions.htm) | With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function. |
| [The this pointer in C++](http://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm) | Every object has a special pointer **this** which points to the object itself. |
| [Pointer to C++ classes](http://www.tutorialspoint.com/cplusplus/cpp_pointer_to_class.htm) | A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it. |
| [Static members of a class](http://www.tutorialspoint.com/cplusplus/cpp_static_members.htm) | Both data members and function members of a class can be declared as static. |

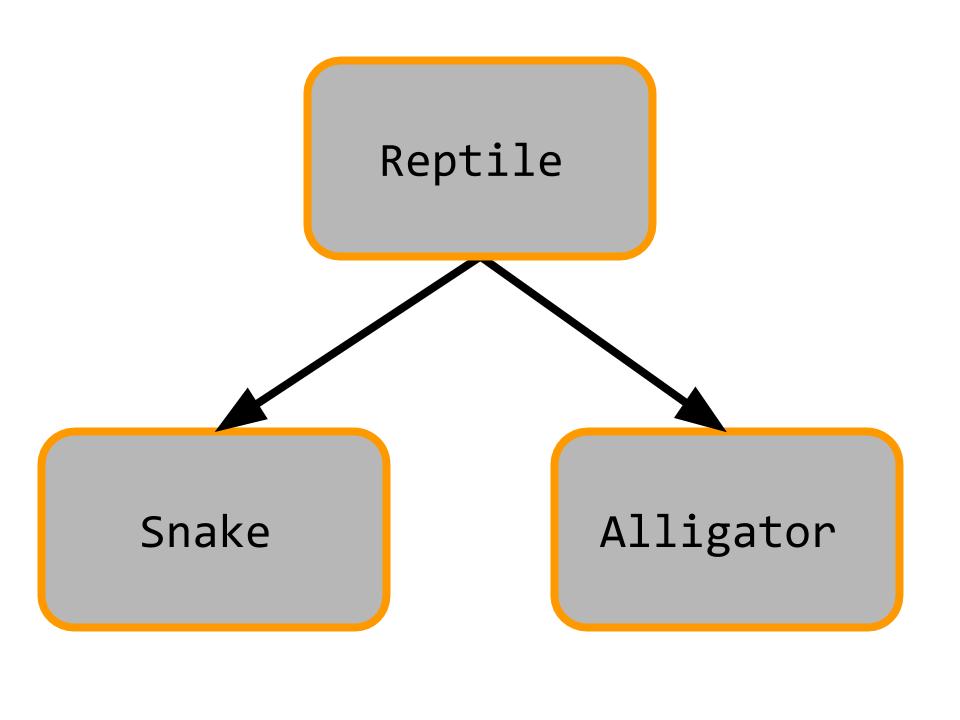
**Encapsulation vs friend function:**

1. Friend function access with private data of a class which break the concept of data hiding of encapsulation.
2. In the extended sense basically friendship concept in oop extend encapsulation. ☺ example:
3. class X {
4. int i;
5. public:
6. void m(); // grant X::m() access
7. friend void f(X&); // grant f(X&) access
8. // ...
9. };
10. void X::m() { i++; /\* X::m() can access X::i \*/ }
11. void f(X& x) { x.i++; /\* f(X&) can access X::i \*/ }

**Inheritance vs Composition:**

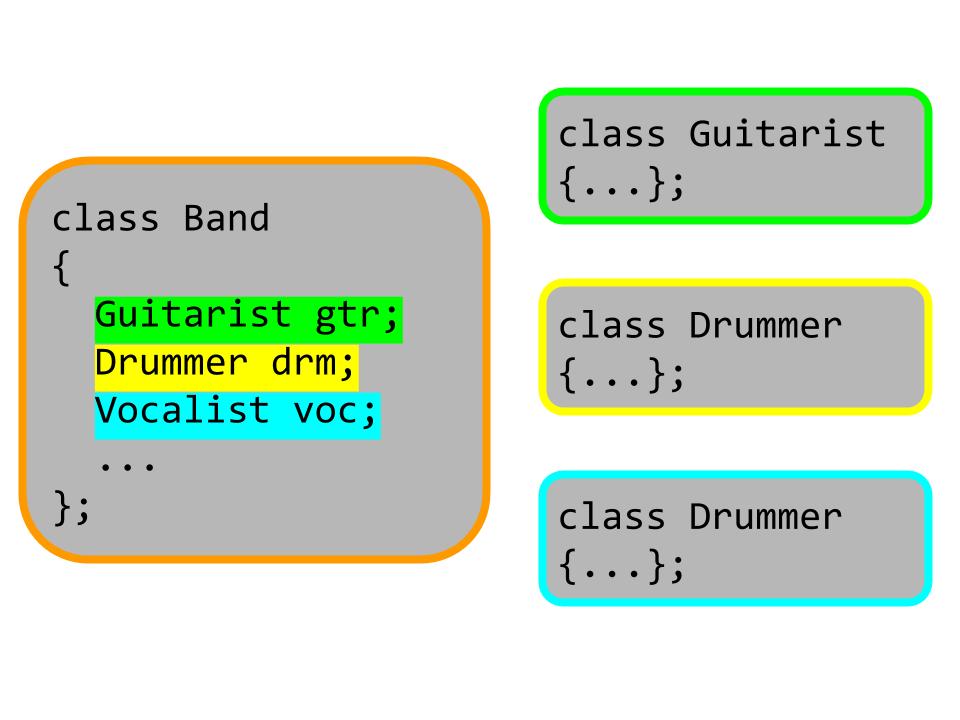
### What is Inheritance?

Inheritance is the process of passing the attributes and behaviors of one class down to descendant classes. For example, you can start with a class named Reptile and have its descendants, Snake and Alligator, inherit the data and members of the Reptile class. Although the Snake and Alligator classes have their own particular attributes and behaviors, each class is derived from the same Reptile class.

[](http://69.195.124.245/~pumpkip1/wp-content/uploads/2014/07/is_has-5.jpg)

### What is Composition

Composition is the process of making one class a data member of another class. For example, you could have a class called Band. The data members of Band could consist of objects from the Guitarist, Drummer, and Vocalist classes. These objects are data members of the Band class, but not descendants or parent classes. They are related by composition, not by inheritance.

[](http://69.195.124.245/~pumpkip1/wp-content/uploads/2014/07/is_has-4.jpg)

**How to prevent Inheritance?**

An example:

**#include <iostream>**

**using namespace std;**

**class Cbase {**

**friend class Cderived; //only friend class can inherit Cbase. As its c'tor is private..**

**friend class Chello;**

**private:**

**Cbase () { //private constructor**

**}**

**};**

**//Cbase become direct base of Cderived..**

**class Cderived : public virtual Cbase{**

**public:**

**Cderived () {**

**}**

**};**

**class Chello : public Cbase { //prevents inheritance**

**public:**

**Chello () { //it can not call Cbase's constructor..**

**}**

**};**

**int main()**

**{**

**Chello ch;**

**Cderived cd;**

**return 0;**

**}**

**Problem with multiple inheritance:**

Taking a look at the graphics below helps in explaining the diamond problem.

|  |
| --- |
| http://www.programmerinterview.com/images/Diamond_inheritance.png |

|  |
| --- |
|  |

Suppose we have 2 classes B and C that derive from the **same** class – in our example above it would be class A. We also have class D that derives from **both** B and C by using multiple inheritance. You can see in the figure above that the classes essentially form the shape of a diamond – which is why this problem is called the diamond problem. Now, let’s take the graphic above and make it more concrete by translating it into actual code:

|  |
| --- |
| /\*  The Animal class below corresponds to class  A in our graphic above  \*/    class Animal { /\* ... \*/ }; // base class  {  int weight;  public:  int getWeight() { return weight;};  };  class Tiger : public Animal { /\* ... \*/ };  class Lion : public Animal { /\* ... \*/ }    class Liger : public Tiger, public Lion { /\* ... \*/ }; |

In the code above, we’ve given a more concrete example of the diamond problem. The Animal class corresponds to the topmost class in the hierarchy (A in our graphic above), Tiger and Lion respectively correspond to B and C in the graphic, and the Liger class corresponds to D.

Now, the question is what is the problem with having an inheritance hierarcy like this. Take a look at the code below so that we can best answer that question:

|  |
| --- |
| int main( )  {  Liger lg ;  /\*COMPILE ERROR, the code below will not get past  any C++ compiler \*/  int weight = lg.getWeight();  } |

|  |
| --- |
|  |

In our inheritance hierarchy, we can see that both the Tiger and Lion classes derive from the Animal base class. And here is the problem: **because Liger derives from both the Tiger and Lion classes – which each have their own copy of the data members and methods of the Animal class- the Liger object "lg" will contain two subobjects of the Animal base class.**

So, you ask, what’s the problem with a Liger object having 2 **sub**-objects of the Animal class? Take another look at the code above – the call "lg.getWeight()" will result in a compiler error. This is because the compiler **does not know whether the call to getWeight refers to the copy of getWeight that the Liger object lg inherited through the Lion class or the copy that lg inherited through the Tiger class**. So, the call to getWeight in the code above is ambiguous and will not get past the compiler.

## Solution to the Diamond Problem

We’ve given an explanation of the diamond problem, but now we want to give you a solution to the diamond problem. If the inheritance from the Animal class to both the Lion class and the Tiger class is marked as virtual, then C++ will ensure that only one subobject of the Animal class will be created for every Liger object. This is what the code for that would look like:

|  |
| --- |
| class Tiger : virtual public Animal { /\* ... \*/ };  class Lion : virtual public Animal { /\* ... \*/ } |

You can see that the only change we’ve made is to add the "virtual" keyword to the Tiger and Lion class declarations. Now the Liger class object will have only one Animal subobject, and the code below will compile just fine:

|  |
| --- |
| int main( )  {  Liger lg ;  /\*THIS CODE WILL NOW COMPILE OK NOW THAT WE'VE  USED THE VIRTUAL KEYWORD IN THE TIGER AND LION  CLASS DECLARATIONS \*/  int weight = lg.getWeight();  } |

**Compile-time and runtime polymorphism:**

## Compile time Polymorphism:

C++ support polymorphism. One function multiple purpose, or in short many functions having same name but with different function body.  
For every function call compiler binds the call to one function definition at compile time. This decision of binding among several functions is taken by considering formal arguments of the function, their data type and their sequence.

### Example of compile time polymorphism:

Example 1: [example of compile time polymorphism; static time binding](http://www.questionscompiled.com/answer/cpp/99/example-of-function-overloading-static-compile-time-polymorphism)

void f(int i){cout<<"int";}

void f(char c){cout<<"char";}

int main()  
{  
   f(10);  
   return 0;  
}

Output: int

## Run time polymorphism:

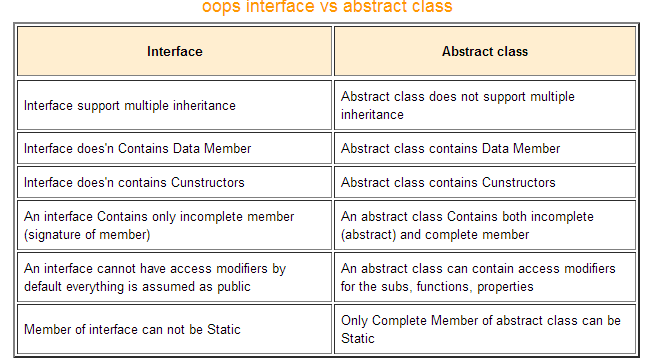
C++ allows binding to be delayed till run time. When you have a function with same name, equal number of arguments and same data type in same sequence in base class as well derived class and a function call of form: base\_class\_type\_ptr->member\_function(args); will always call base class member function. The keyword ***virtual*** on a member function in base class indicates to the compiler to delay the binding till run time.  
  
Every class with atleast one virtual function has a [**vtable**](http://www.questionscompiled.com/answer/cpp/189/vtable-virtual-table-in-cpp) that helps in binding at run time. Looking at the content of base class type pointer it will correctly call the member function of one of possible derived / base class member function.

### Example of run time polymorphism:

class Base  
{  
public:  
  
    virtual void display(int i)  
    { cout<<"Base::"<<i; }  
};  
  
class Derv: public Base  
{  
public:  
  
    void display(int j)  
    { cout<<"Derv::"<<j; }  
};  
  
int main()  
{  
    Base \*ptr=new Derv;  
    ptr->display(10);  
    return 0;  
}

Output: Derv::10

**Interfaces and abstract classes:**



**Interfaces & abstract classes:**

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

class Box

{

public:

// pure virtual function

virtual double getVolume() = 0;

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called **concrete classes**.

## Abstract Class Example:

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()**:

#include <iostream>

using namespace std;

// Base class

class Shape

{

public:

// pure virtual function providing interface framework.

virtual int getArea() = 0;

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// Derived classes

class Rectangle: public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

class Triangle: public Shape

{

public:

int getArea()

{

return (width \* height)/2;

}

};

int main(void)

{

Rectangle Rect;

Triangle Tri;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total Rectangle area: " << Rect.getArea() << endl;

Tri.setWidth(5);

Tri.setHeight(7);

// Print the area of the object.

cout << "Total Triangle area: " << Tri.getArea() << endl;

return 0;

}

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

Total Rectangle area: 35

Total Triangle area: 17

You can see how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.

## What is a virtual base class?

- An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.  
- C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

## What is Virtual base class? Explain its uses.

- When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class’ name with the word virtual.  
- Consider the following example :

class A   
{   
   public:   
       int i;   
};  
  
class B : virtual public A   
{   
   public:   
       int j;   
};  
  
class C: virtual public A   
{   
   public:   
       int k;   
};  
  
class D: public B, public C   
{   
   public:   
       int sum;   
};  
  
int main()   
{   
   D ob;   
   ob.i = 10; //unambiguous since only one copy of i is inherited.   
   ob.j = 20;   
   ob.k = 30;   
   ob.sum = ob.i + ob.j + ob.k;   
   cout << “Value of i is : ”<< ob.i<<”\n”;   
   cout << “Value of j is : ”<< ob.j<<”\n”; cout << “Value of k is :”<< ob.k<<”\n”;   
   cout << “Sum is : ”<< ob.sum <<”\n”;   
  
   return 0;   
}

**Instantiating derived class with base class constructor:**

class BaseClass {

public:

BaseClass (string a);

};

class DerivedClass : public BaseClass {

public:

DerivedClass (string b);

};

int main() {

DerivedClass abc ("Hello");

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class DerivedClass : public BaseClass {

public:

DerivedClass (string b);

};

/\* ... \*/

DerivedClass::DerivedClass(string b) : BaseClass(b)

{}

class DerivedClass : public BaseClass {

public:

DerivedClass(int a, string b, string c);

private:

int x;

};

DerivedClass::DerivedClass(int a, string b, string c) : BaseClass(b + c), x(a)

{}

**Friend of Multiple classes:**

Yes it is possible. An example will explain this best:

class B; //defined later

void add(A,B);

class A{

private:

int a;

public:

A(){

a = 100;

}

friend void add(A,B);

};

class B{

private:

int b;

public:

B(){

b = 100;

}

friend void add(A,B);

};

void add (A Aobj, B Bobj){

cout << (Aobj.a + Bobj.b);

}

main(){

A A1;

B B1;

add(A1,B1);

return 0;

}

Document Created by

**Md. Abdul Mazed**

CSTE 9th Batch

Noakhali Science & Technology University