**Lesson 19: Inheritance in C++**

The ability to use the object-oriented programming is an important feature of C++. [Lesson 12: classes in C++](http://www.cprogramming.com/tutorial/lesson12.html) introduced the idea of the class; if you have not read it and do not know the basic details of classes, you should read it before continuing this tutorial.   
  
Inheritance is an important feature of classes; in fact, it is integral to the idea of object oriented programming. Inheritance allows you to create a hierarchy of classes, with various classes of more specific natures inheriting the general aspects of more generalized classes. In this way, it is possible to structure a program starting with abstract ideas that are then implemented by specific classes. For example, you might have a class Animal from which class dog and cat inherent the traits that are general to all animals; at the same time, each of those classes will have attributes specific to the animal dog or cat.

Inheritance offers many useful features to programmers. The ability, for example, of a variable of a more general class to function as any of the more specific classes which inherit from it, called polymorphism, is handy. For now, we will concentrate on the basic syntax of inheritance. Polymorphism will be covered in its own tutorial.  
  
Any class can inherit from any other class, but it is not necessarily good practice to use inheritance (put it in the bank rather than go on a vacation). Inheritance should be used when you have a more general class of objects that describes a set of objects. The features of every element of that set (of every object that is also of the more general type) should be reflected in the more general class. This class is called the base class. base classes usually contain functions that all the classes inheriting from it, known as derived classes, will need. base classes should also have all the variables that every derived class would otherwise contain.  
  
Let us look at an example of how to structure a program with several classes. Take a program used to simulate the interaction between types of organisms, trees, birds, bears, and other creatures coinhabiting a forest. There would likely be several base classes that would then have derived classes specific to individual animal types. In fact, if you know anything about biology, you might wish to structure your classes to take advantage of the biological classification from Kingdom to species, although it would probably be overly complex. Instead, you might have base classes for the animals and the plants. If you wanted to use more base classes (a class can be both a derived of one class and a base of another), you might have classes for flying animals and land animals, and perhaps trees and scrub. Then you would want classes for specific types of animals: pigeons and vultures, bears and lions, and specific types of plants: oak and pine, grass and flower. These are unlikely to live together in the same area, but the idea is essentially there: more specific classes ought to inherit from less specific classes.  
  
Classes, of course, share data. A derived class has access to most of the functions and variables of the base class. There are, however, ways to keep a derived class from accessing some attributes of its base class. The keywords public, protected, and private are used to control access to information within a class. It is important to remember that public, protected, and private control information both for specific instances of classes and for classes as general data types. Variables and functions designated public are both inheritable by derived classes and accessible to outside functions and code when they are elements of a specific instance of a class. Protected variables are not accessible by functions and code outside the class, but derived classes inherit these functions and variables as part of their own class. Private variables are neither accessible outside the class when it is a specific class nor are available to derived classes. Private variables are useful when you have variables that make sense in the context of large idea.

**Lesson 20: C++ Inheritance - Syntax**

Before beginning this lesson, you should have an understanding of the idea of inheritance. If you do not, please read [lesson 19](http://www.cprogramming.com/tutorial/lesson19.html). This lesson will consist of an overview of the syntax of inheritance, the use of the keywords public, private, and protected, and then an example program following to demonstrate each.

The syntax to denote one class as inheriting from another is simple. It looks like the following: class Bear : public Animal, in place of simply the keyword class and then the class name. The ": public *base\_class\_name*" is the essential syntax of inheritance; the function of this syntax is that the class will contain all public and protected variables of the base class. Do not confuse the idea of a derived class having access to data members of a base class and specific instances of the derived class possessing data. The data members - variables and functions - possessed by the derived class are specific to the type of class, not to each individual object of that type. So, two different Bear objects, while having the same member variables and functions, may have different information stored in their variables; furthermore, if there is a class Animal with an object, say object BigAnimal, of that type, and not of a more specific type inherited from that class, those two bears will not have access to the data within BigAnimal. They will simply possess variables and functions with the same name and of the same type.   
  
A quick example of inheritance:

class Animal

{

public:

Animal();

~Animal();

void eat();

void sleep();

void drink();

private:

int legs;

int arms;

int age;

};

//The class Animal contains information and functions

//related to all animals (at least, all animals this lesson uses)

class Cat : public Animal

{

public:

int fur\_color;

void purr();

void fish();

void markTerritory();

};

//each of the above operations is unique

//to your friendly furry friends

//(or enemies, as the case may be)

A discussion of the keywords public, private, and protected is useful when discussing inheritance. The three keywords are used to control access to functions and variables stored within a class.

#### public:

The most open level of *data hiding* is public. Anything that is public is available to all derived classes of a base class, and the public variables and data for each object of both the base and derived class is accessible by code outside the class. Functions marked public are generally those the class uses to give information to and take information from the outside world; they are typically the interface with the class. The rest of the class should be hidden from the user using private or protected data (This hidden nature and the highly focused nature of classes is known collectively as encapsulation). The syntax for public is:

public:

Everything following is public until the end of the class or another data hiding keyword is used.   
  
In general, a well-designed class will have no public fields--everything should go through the class's functions. Functions that retrieve variables are known as 'getters' and those that change values are known as 'setters'. Since the public part of the class is intended for use by others, it is often sensible to put the public section at the top of the class.

#### protected:

Variables and functions marked protected are inherited by derived classes; however, these derived classes hide the data from code outside of any instance of the object. Keep in mind, even if you have another object of the same type as your first object, the second object cannot access a protected variable in the first object. Instead, the second object will have its own variable with the same name - but not necessarily the same data. Protected is a useful level of access control for important aspects to a class that must be passed on without allowing it to be accessed. The syntax is the same as that of public. specifically,

protected:

#### private:

Private is the highest level of data-hiding. Not only are the functions and variables marked private not accessible by code outside the specific object in which that data appears, but private variables and functions are not inherited (in the sense that the derived class cannot directly access these variables or functions). The level of data protection afforded by protected is generally more flexible than that of the private level. On the other hand, if you do not wish derived classes to access a method, declaring it private is sensible.

private:

# Class Design in C++

## Understanding Interfaces

When you're designing a class in C++, the first thing you should decide is the public interface for the class. The public interface determines how your class will be used by other programmers (or you), and once designed and implemented it should generally stay pretty constant. You may decide to add to the interface, but once you've started using the class, it will be hard to remove functions from the public interface (unless they aren't used and weren't necessary in the first place).

But that doesn't mean that you should include more functionality in your class than necessary just so that you can later decide what to remove from the interface. If you do this, you'll just make the class harder to use. People will ask questions like, "why are there four ways of doing this? Which one is better? How can I choose between them?" It's usually easier to keep things simple and provide one way of doing each thing unless there's a compelling reason why your class should offer multiple methods with the same basic functionality.   
  
At the same time, just because adding methods to the public interface (probably) won't break anything that doesn't mean that you should start off with a tiny interface. First of all, if anybody decides to inherit from your class and you then choose a function with the same name, you're in for a boatload of confusion. First, if you don't declare the function virtual, then an object of the subclass will have the function chosen depending on the static type of the pointer. This can be messy. Moreover, if you do declare it virtual, then you have the issue that it might provide a different type of functionality than was intended by the original implementation of that function. Finally, you just can't add a pure virtual function to a class that's already in use because nobody who has inherited from it will have implemented that function.   
  
The public interface, then, should remain as constant as possible. In fact, a good approach to designing classes is to write the interface before the implementation because it's what determines how your class interacts with the rest of the world (which is more important for the program as a whole than how the class is actually implemented). Moreover, if you write the interface first, you can get a feel for how the class will work with other classes before you actually dive into the implementation details.

## Inheritance and Class Design

The second issue of your class design is what should be available to programmers who wish to create subclasses. This interface is primarily determined by virtual functions, but you can also include protected methods that are designed for use by the class or its subclasses (remember that protected methods are visible to subclasses while private methods are not).   
  
A key consideration is whether it makes sense for a function to be virtual. A function should be virtual when the implementation is likely to differ from subclass to subclass. Vice-versa, whenever a function should not change, then it should be made non-virtual. The key idea is to think about whether to make a function virtual by asking if the function should always be the same for every class.   
  
For example, if you have a class is designed to allow users to monitor network traffic and you want to allow subclasses that implement different ways of analyzing the traffic, you might use the following interface:

class TrafficWatch

{

public:

// Packet is some class that implements information about network

// packets

void addPacket (const Packet& network\_packet);

int getAveragePacketSize ();

int getMaxPacket ();

virtual bool isOverloaded ();

};

In this class, some methods will not change from implementation to implementation; adding a packet should always be handled the same way, and the average packet size isn't going to change either. On the other hand, someone might have a very different idea of what it means to have an overloaded network. This will change from situation to situation and we don't want to prevent someone from changing how this is computed--for some, anything over 10 Mbits/sec of traffic might be an overloaded network, and for others, it would require 100 Mbits/sec on some specific network cables.   
  
Finally, when publicly inheriting from any class or designing for inheritance, remember that you should strive for it to be clear that inheritance models is-a. At heart, the is-a relationship means that the subclass should be able to appear anywhere the parent class could appear. From the standpoint of the user of the class, it should not matter whether a class is the parent class or a subclass.

To design an is-a relationship, make sure that it makes sense for the class to include certain functions to be sure that it doesn't include that subclasses might not actually need. One example of having an extra function is that of a Bird class that implements a fly function. The problem is that not all birds can fly--penguins and emus, for instance. This suggests that a more prudent design choice might be to have two subclasses of birds, one for birds that can fly and one for flightless birds. Of course, it might be overkill to have two subclasses of bird depending on how complex your class hierarchy will be. If you know that nobody would ever expect use your class for a flightless bird, then it's not so bad. Of course, you won't always know what someone will use your class for and it's much easier to think carefully before you start to implement an entire class hierarchy than it will be to go back and change it once people are using it.

# Understanding Initialization Lists in C++

## nderstanding the Start of an Object's Lifetime

In C++, whenever an object of a class is created, its constructor is called. But that's not all--its parent class constructor is called, as are the constructors for all objects that belong to the class. By default, the constructors invoked are the default ("no-argument") constructors. Moreover, all of these constructors are called before the class's own constructor is called.

For instance, take the following code:

#include <iostream>

class Foo

{

public:

Foo() { std::cout << "Foo's constructor" << std::endl; }

};

class Bar : public Foo

{

public:

Bar() { std::cout << "Bar's constructor" << std::endl; }

};

int main()

{

// a lovely elephant ;)

Bar bar;

}

The object bar is constructed in two stages: first, the Foo constructor is invoked and then the Bar constructor is invoked. The output of the above program will be to indicate that Foo's constructor is called first, followed by Bar's constructor.   
  
Why do this? There are a few reasons. First, each class should need to initialize things that belong to it, not things that belong to other classes. So a child class should hand off the work of constructing the portion of it that belongs to the parent class. Second, the child class may depend on these fields when initializing its own fields; therefore, the constructor needs to be called before the child class's constructor runs. In addition, all of the objects that belong to the class should be initialized so that the constructor can use *them* if it needs to.   
  
But what if you have a parent class that needs to take arguments to its constructor? This is where initialization lists come into play. An initialization list immediately follows the constructor's signature, separated by a colon:

class Foo : public parent\_class

{

Foo() : parent\_class( "arg" ) // sample initialization list

{

// you must include a body, even if it's merely empty

}

};

Note that to call a particular parent class constructor, you just need to use the name of the class (it's as though you're making a function call to the constructor).   
  
For instance, in our above example, if Foo's constructor took an integer as an argument, we could do this:

#include <iostream>

class Foo

{

public:

Foo( int x )

{

std::cout << "Foo's constructor "

<< "called with "

<< x

<< std::endl;

}

};

class Bar : public Foo

{

public:

Bar() : Foo( 10 ) // construct the Foo part of Bar

{

std::cout << "Bar's constructor" << std::endl;

}

};

int main()

{

Bar stool;

}

## Using Initialization Lists to Initialize Fields

In addition to letting you pick which constructor of the parent class gets called, the initialization list also lets you specify which constructor gets called for the objects that are fields of the class. For instance, if you have a string inside your class:

class Qux

{

public:

Qux() : \_foo( "initialize foo to this!" ) { }

// This is nearly equivalent to

// Qux() { \_foo = "initialize foo to this!"; }

// but without the extra call to construct an empty string

private:

std::string \_foo;

};

Here, the constructor is invoked by giving the name of the object to be constructed rather than the name of the class (as in the case of using initialization lists to call the parent class's constructor).   
  
If you have multiple fields of a class, then the names of the objects being initialized should appear in the order they are declared in the class (and after any parent class constructor call):

class Baz

{

public:

Baz() : \_foo( "initialize foo first" ), \_bar( "then bar" ) { }

private:

std::string \_foo;

std::string \_bar;

};

### Initialization Lists and Scope Issues

If you have a field of your class that is the same name as the argument to your constructor, then the initialization list "does the right thing." For instance,

class Baz

{

public:

Baz( std::string foo ) : foo( foo ) { }

private:

std::string foo;

};

is roughly equivalent to

class Baz

{

public:

Baz( std::string foo )

{

this->foo = foo;

}

private:

std::string foo;

};

That is, the compiler knows which foo belongs to the object, and which foo belongs to the function.

### Initialization Lists and Primitive Types

It turns out that initialization lists work to initialize both user-defined types (objects of classes) and primitive types (e.g., int). When the field is a primitive type, giving it an argument is equivalent to assignment. For instance,

class Quux

{

public:

Quux() : \_my\_int( 5 ) // sets \_my\_int to 5

{ }

private:

int \_my\_int;

};

This behavior allows you to specify templates where the templated type can be either a class or a primitive type (otherwise, you would have to have different ways of handling initializing fields of the templated type for the case of classes and objects).

template <class T>

class my\_template

{

public:

// works as long as T has a copy constructor

my\_template( T bar ) : \_bar( bar ) { }

private:

T \_bar;

};

### Initialization Lists and Const Fields

Using initialization lists to initialize fields is not always necessary (although it is probably more convenient than other approaches). But it is necessary for [const](http://www.cprogramming.com/tutorial/const_correctness.html) fields. If you have a const field, then it can be initialized only once, so it must be initialized in the initialization list.

class const\_field

{

public:

const\_field() : \_constant( 1 ) { }

// this is an error: const\_field() { \_constant = 1; }

private:

const int \_constant;

};

### When Else do you Need Initialization Lists?

#### No Default Constructor

If you have a field that has no default constructor (or a parent class with no default constructor), you must specify which constructor you wish to use.

#### References

If you have a field that is a reference, you also must initialize it in the initialization list; since references are immutable they can be initialized only once.

## Initialization Lists and Exceptions

Since constructors can throw exceptions, it's possible that you might want to be able to handle exceptions that are thrown by constructors invoked as part of the initialization list.   
  
First, you should know that even if you catch the exception, it will get rethrown because it cannot be guaranteed that your object is in a valid state because one of its fields (or parts of its parent class) couldn't be initialized. That said, one reason you'd want to catch an exception here is that there's some kind of translation of error messages that needs to be done.   
  
The syntax for catching an exception in an initialization list is somewhat awkward: the 'try' goes right before the colon, and the catch goes after the body of the function:

class Foo

{

Foo() try : \_str( "text of string" )

{

}

catch ( ... )

{

std::cerr << "Couldn't create \_str";

// now, the exception is rethrown as if we'd written

// "throw;" here

}

};

## Initialization Lists: Summary

Before the body of the constructor is run, all of the constructors for its parent class and then for its fields are invoked. By default, the no-argument constructors are invoked. Initialization lists allow you to choose which constructor is called and what arguments that constructor receives.

If you have a reference or a const field, or if one of the classes used does not have a default constructor, you must use an initialization list.