A Tutorial

Object-Oriented Programming with C++

This material was originally written for the book *Object-Oriented Programming with C++ and OSF/Motif*, by Douglas Young, Prentice Hall, 1992. It was cut from that text because it seemed slightly off the central topic of that book, which is how to use object-oriented techniques to write X and Motif applications. Rough edges, due to its extraction in a somewhat incomplete state from the book for which it was originally written, undoubtedly exist. Hopefully, it may be useful to some as a tutorial introduction to object-oriented programming.

This tutorial introduces the fundamental concepts of object-oriented programming and shows how C++ supports this technique. Section 1 contrasts object-oriented programming with more traditional techniques and provides an overview of the characteristics and advantages of the object-oriented approach. Section 2 discusses individual aspects of object-oriented programming, with an emphasis on the related features of C++. Section 3 presents some of the features of C++ that, while not directly related to object-oriented programming, are particularly useful in conjunction with an object-oriented style.

1 What Is Object-Oriented Programming?

Most software developers realize that the process of writing software is too complex to allow applications to be written as a single, continuous stream of instructions. Beginning programmers soon learn to break large programs into smaller components, often called subroutines, functions, procedures, or modules. Each of these components corresponds to some subset of the entire task performed by the program. Therefore, one of the first questions that must be answered before beginning any significant programming effort is, “how can the problem be split into smaller problems?” Once this question is answered, the resulting smaller problems become the modules or subcomponents of the larger program.

This technique of solving large or complex problems by breaking a problem down into smaller pieces that can be solved individually is called *decomposition*. The main difference between object-oriented programming and more traditional approaches lies in the way in which problems are decomposed. The traditional, non-object-oriented approach to decomposing problems focuses on the operations or activities that need to be performed. The programmer begins by breaking a problem into several tasks. These tasks are then divided into sub-tasks, which can be broken down even further. At some point, each task becomes small enough that it is practical to implement it as a relatively small function.

The function-oriented approach is undoubtedly familiar to all programmers, but let’s look at a simple example so we can contrast it with an object-oriented approach. The problem to be solved is the design and implementation of a first-in-last-out (FILO) queue, also known as a *stack*. This problem can be split into two parts: placing data onto the stack, and removing data from the stack.

As typically implemented, the stack is a data structure of some type (perhaps an array), declared and allocated somewhere in the program. A pair of functions, `push()` and `pop()`, perform the two basic tasks mentioned above. The stack data structure is passed as an argument to the `push()` and `pop()` functions, which modify the stack by adding or removing items. Figure 1 shows a data flow diagram of a program using a functional implementation of a stack. In this figure, the circles represent procedures and the arrows represent the flow of the data named beside the arrows.

In this scenario, it is the responsibility of the programmer who uses these functions to ensure that all parts of the program use the correct data structure to represent the stack. If the programmer wishes to use a different representation for the stack (perhaps a linked list instead of an array), he or she must also change all parts of the program that refer to the stack. When using traditional programming techniques, the data and the functions that operate on that data tend to be distributed throughout a program. A change to one part of a system is likely to require changes to many other parts as well. In the example in Figure 1, the main body of the program and both the `pop()` and `push()` functions would have to be modified to support any new representation.

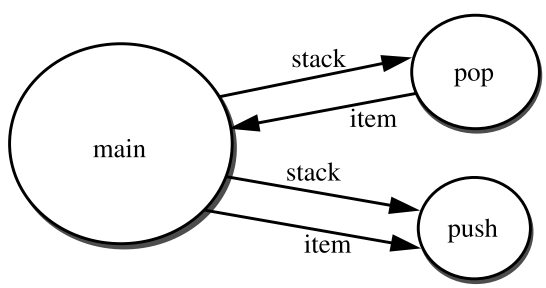

Figure 1 Using a procedural stack implementation.

Instead of focusing on the tasks to be performed, object-oriented programmers design software around the *objects* in a system. In an object-oriented program, a stack would undoubtedly be implemented as an object. The stack object would be a self-contained entity that supports two operations named `push` and `pop`. The actual data structure used to represent the stack would be hidden inside the object as would the implementation of the `push` and `pop` operations. Applications would interact with the stack only by sending `push` and `pop` *messages* to the stack object, not by passing the stack to separate `push()` and `pop()` functions. Instead of viewing a stack as three distinct entities (two functions and a data structure), object-oriented programmers view a stack as a single entity that can handle several messages.

Figure 2 represents an object-oriented implementation of a stack. Here, the circle represents a procedure (`main`), while the rectangle represents a stack object. The object has two input ports, labeled `push` and `pop`. The arrows connected to these input areas represent a message sent to the object, and also show the flow of data. Here, the data includes the items pushed or popped from the stack.

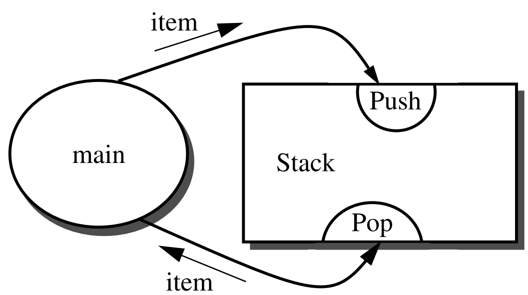

Figure 2 Using a stack object.

To the outside world, the stack object is a "black box." The only thing we know about it is that we can push data onto the stack, and pop the data back off. This stack object could be replaced with any other stack object that has the same external interface. Applications that use one stack implementation should continue to work, even if the original stack is replaced by another one that has the same external interface but a completely different internal implementation.

The difference between these two approaches has some important consequences. A Stack object completely hides the data used to represent the stack from the other parts of the program and also hides the implementation of the operations on the stack. The stack offers a well-defined external interface, provided by the operations `push` and `pop`. As long as that interface remains constant, the programmer is free to change the internal data structure used to represent the stack, or the implementation of the operations on that data. When using object-oriented techniques, all data and the functions that operate on that data are contained in one location, instead of being distributed throughout an application. Therefore, the implementation of individual objects can be changed without affecting other parts of a system.

Let's assume that a programmer decides to use an existing stack object from a library, which is probably written and maintained by someone else. Now, suppose that the programmer who supports that library implements a faster, more robust version of a stack by changing the data structure used internally by the stack. Changing the data structure will most likely also require the `push` and `pop` operations to be reimplemented as well. However, the external interface to `push` and `pop` should not have to change. Because such changes only affect the internal implementation and not the external interface to the stack, programmers who previously used the older version can simply relink with the new library and receive the benefits of the new stack, without changing any code.

This is far less likely to be the case with a procedural stack library package, such as that described in the previous section, because the non-object-oriented approach is more likely to expose internal details, such as the precise definition of the stack data structure. Objects *encapsulate* data and operations on the object's data to insulate applications from the internal implementation details of that object. This does not mean, of course, that a careful programmer can't design and implement highly encapsulated data structures in non-object-oriented languages. It is also possible to write poorly encapsulated data structures in most object-oriented languages. The difference is in the degree of support various languages provide for encapsulation and data abstraction.

So far, we have not stated specifically what object-oriented programming is. Unfortunately, it is difficult to develop a precise definition of object-oriented programming that everyone can agree on, although many people have tried. In fact, there are many, often conflicting, definitions. The previous sections hinted at one definition, which can be stated as follows:

> *Object-oriented programming is a technique that emphasizes the objects in a system rather than the tasks the system performs.*

This simple idea is the heart of the object-oriented approach, but it leaves many details to the imagination. The previous discussion concentrates on one element of object-oriented programming, often called *data abstraction*. Data abstraction is an important characteristic of object-oriented programming, but it is not the only one. The following are some features typically associated with object-oriented programming:

- An *object* is a cohesive package of data and functions that operate on that data.
- An object is an instance of a *class*, which defines the structure and behavior of all objects that belong to the class.
- Some or all of an object's data can be specified as private to the object, such that this data cannot be accessed by any part of a program outside the object. This is referred to as *encapsulation* or *data abstraction*.
- The operations supported by an object are often referred to as *methods*. Methods are invoked by sending a *message* to an object.
- *Polymorphism* allows different types of objects to respond to the same message in different ways, without requiring the program to know the object's exact type.
- *Inheritance* allows classes to share the behavior of other classes.

One reason object-oriented programming is difficult to define is that "object-oriented" is really just a way of thinking. It is an attitude; an approach to designing a system. The specific details listed above are just a few features of various languages that programmers can use to implement an object-oriented design.

Notice that programmers can use almost any language to develop software using object-oriented techniques. The Xt Intrinsics library, on which Motif is based, provides an example of a system that implements most of the characteristics listed above in ordinary C. However, an object-oriented language can make the task much easier for the programmer. While it is possible, developing an object-oriented system in C requires a great deal more effort than when using a language like C++, which directly supports object-oriented programming. The following sections examine each of the characteristics listed above in more detail, and demonstrate how C++ supports these object-oriented techniques.

The Benefits of Object-oriented Programming

The previous section discusses some of the benefits of data abstractions. However, object-oriented programming offers other advantages, as well. Object-oriented techniques can benefit software developers in several ways, which include:

1. *Reusability.* Objects provide a way to develop software that can be used in multiple parts of a program, or even by multiple projects. Well-designed classes are often largely self-contained and have few external dependencies. This allows programmers to treat objects as components that can be plugged in wherever they are needed. Brad Cox [Cox86] stresses the idea that objects are the software equivalent of an integrated circuit that can be plugged into many different circuit boards. Cox refers to this type of object as a *Software-IC* to emphasize the analogy. The promise of reusability is one of the greatest attractions of object-oriented programming. If a software component can be written once but used many times, programmers will be more productive and programs will be more robust.

2. *Maintainability.* Even when a class is used only once, the emphasis object-oriented programming places on clean interfaces between self-contained software modules is an important benefit in itself. Because objects are self-contained, changes made to any particular object are less likely to affect other objects in the system. Maintenance typically consumes a far greater part of the software lifecycle than initial development, so any technique that can make software easier to maintain is worth exploring.

3. *Rapid-prototyping and development*. Inheritance allows programmers to create new types of objects by extending and altering existing types. This often allows programmers to prototype complex systems with surprising speed.

4. *Extensibility.* Most object-oriented languages allow programmers to declare types of objects as new data types. These types allow programmers to effectively extend the language. Also, most object-oriented languages provide mechanisms that allow programmers to extend the capabilities of existing types of objects.

5. *Conceptual consistency.* In many cases, objects within a program correspond directly to objects in the real world. In traditional programming, programmers have to construct a complex mapping between the abstract model of what the program does and the functions and procedures that implement that model. In object-oriented systems, the objects often correspond directly to the abstractions modeled by the program, which can make programs easier to understand.

The extent to which these claimed benefits of object-oriented programming have a measurable impact on real software development is still open to debate. Solid evidence is hard to find or substantiate. Those who believe in object-oriented techniques often claim phenomenal improvements in productivity, quality of code, and maintainability later in the software lifecycle. Others point to projects that used object-oriented techniques and failed. In either case, it is seldom clear whether object-oriented programming was the key to the claimed success or failure, or whether other more significant factors affected the final outcome.

In spite of the lack of solid evidence, object-oriented programming is widely recognized as a valuable tool, applicable to a wide variety of situations. Programmers who learn to apply object-oriented techniques seldom return willingly to their earlier programming styles.

2 The Elements of Object-oriented Programming

Section 1 mentions several common characteristics often identified with object-oriented programming. The following sections examine each of these features in more detail. In addition to discussing basic object-oriented concepts, we will see how C++ supports classes, inheritance, and other object-oriented techniques.

Objects

An *object* is a self-contained package that encapsulates data and procedures that operate on the object's data. From an abstract perspective, an object is a "thing" – a programmatic abstraction that often closely models some entity in the real world. From a less abstract viewpoint, an object is simply an instance of a complex data structure that includes both data and functions. Object-oriented languages such as C++ provide convenient ways to declare, implement and use these data structures in an object-oriented style.

The stack described in the previous section is one example of an object, but the list of things that can be modeled as objects is endless. For example, consider an electronic mail program. The program receives mail from a low-level mail service such as the UNIX `sendmail` program, and allows the user to view and manipulate individual messages or groups of messages.

One way to design such a program is to model each incoming mail message as an object. The data in the object might include such things as the text of the message, who the mail is for, who it is from, the date on which it was sent, and so on. This object, which we can call a `mailMessage` object, could support all the operations the user might wish to perform on any individual mail message. For example, the `mailMessage` object might support:

- A `send` operation that causes the message to be sent to its destination
- A `reply` operation that sends a message back to whoever originated the message
- A `forward` operation that sends a copy of the message to someone else
- A `print` operation that sends the text of the message to a printer
- A `view` operation that displays the contents of the message on the screen
- A `delete` operation that deletes the contents of the message, and the object itself.

There may be many other objects in the mail system as well. For example, the program might have an `inbox` object, an `outbox` object, and perhaps other container objects that can store collections of related `mailMessage` objects. Such objects would each maintain collections of objects, and support operations such as:

- An `add` operation that adds a `mailMessage` object to the container
- A `remove` operation that removes a `mailMessage` object from the container
- A `sort` operation that sorts the container's contents in various ways
- A `view` operation that displays a list of the container's contents
- A `print` operation that sends a list of the message objects in the container to a printer.

Another object that could be useful in such a system is a dispatcher object that watches for new mail and routes new messages to the `inbox`. We could call this object a `postMaster` object. The `postMaster` object could also route outgoing messages from the outbox to some external electronic mail service.

Figure 3 shows how these objects work together in a simple mail tool. The `postMaster` object receives messages from the external mail system, and routes the incoming `mailMessage` objects to the user's `inbox` object. This object stores all unread `mailMessage` objects. The user can send view messages, print messages, and so on by sending messages to the `inbox` and the `mailMessage` objects through some suitable user interface.

The user can also create new `mailMessage` objects, through some interface not shown here, and place them in an `outbox` object. From there, they are sent to the `postMaster` object for processing and routing to the appropriate destination.

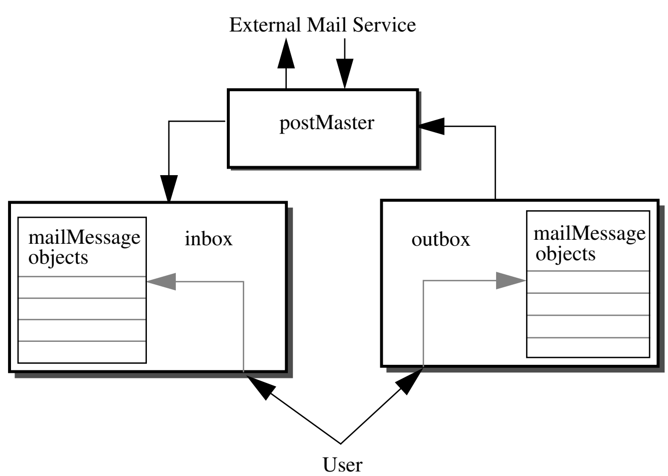

Figure 3 An object-oriented architecture for a mail program.

This example illustrates how a simple system can be viewed as a collection of independent objects. Each object in the mail system is responsible for performing certain tasks and maintaining its own state. Each object interacts with others, but performs its primary task independently.

Classes

In object-oriented programming, the word *class* refers to a category of structurally-identical objects. For example, in the mail program discussed above, all `mailMessage` objects would belong to a class known as the MailMessage class. The MailMessage class would describe the characteristics of all `mailMessage` objects and the operations that could be performed on those objects. When a new mail message arrives, the `postMaster` object creates a unique object to represent that particular message. Each individual object is an *instance* of the MailMessage class. Objects are created by *instantiating* a class.

A class serves as a template for creating objects. The template defines the type of data stored in an object and the operations supported by an object instantiated from that class. However, the actual data stored in each individual `mailMessage` object may vary; each instance is unique. One instance of the MailMessage class may contain a status report to be sent to a co-worker, while another might contain a request for information to be broadcast to a large mailing list.

The various container objects (`inbox`, `outbox`) in the mail system might be different instances of a Container class. Each instance of the Container class in the mail application would then support the same operations, but be used in a different way, and contain different `mailMessage` objects.

Creating Classes in C++

C++ directly supports classes with a new data type, known appropriately as a *class*. The word *class* is a keyword in C++. A C++ class is much like a `struct` in C, but has some additional properties. Let's return to our earlier example and see how a Stack class might be implemented in C++. A very simple C++ class that represents an integer stack can be declared as follows[1]:

```
class Stack {

  protected:

    int  *data;         // Items in the stack
    int   top;          // Index of next open slot

  public:

    Stack();            // Constructor
    ~Stack();           // Destructor
    int  pop();         // Remove an item from stack
    void push ( int ); // Add an item to stack
};
```

This declaration describes a new user-defined data type called Stack that includes the data used to represent the stack and four functions that operate on that data. In addition to the word `class`, this example also contains two other keywords, `protected` and `public`, which will be explained

[1] C programmers: note that C++ uses a double slash (//) to indicate a comment that extends to the end of the line.

as we go on. In many ways, a C++ class behaves like a C structure, but there are several significant differences. Let's look at these, one at a time.

Data Members

Like C structures, C++ classes contain *members*, which can be data or pointers to functions. The Stack class described above contains two *data members*. The first is a pointer to an integer, named `data`, that represents the items on the stack. The second data member, `top` serves as an index that marks the top of the stack. These data members cannot be accessed outside the class because they are declared in the protected portion of the class.

Instantiating C++ Classes

In C++, a class is treated the same as an instance of any other programmer-defined data type. An instance of a class can be created in the same way as any other data structure. Instantiating a class allocates memory for an object. Objects can be static, automatic, or dynamic like other variables. The following example creates two instances of the Stack class as automatic variables:

```
simplefunction()
{
    Stack aStack, anotherStack; // Declare two Stack objects

    // Push one value on each stack

    aStack.push ( 10 );
    anotherStack.push ( 20 );
}
```

This function creates two unique and independent instances of the Stack class. The variables `aStack` and `anotherStack` are automatic variables that exist within the scope of `simple-function()`.

When using an object-oriented programming style, objects are often allocated dynamically on the heap. In this case, the object is usually declared as a pointer and then allocated using the C++ `new` operator. The `new` operator is similar to `malloc()`, normally used in C. However, `new` is the preferred way to allocate memory in C++, and must be used to instantiate classes. Changing the above example to use the `new` operator to allocate the Stack object results in the following code segment:

```
simplefunction()
{
    Stack *aStack       = new Stack(); // Allocate a Stack object
    Stack *anotherStack = new Stack(); // Allocate a Stack object

    // Push one value on each stack

    aStack->push ( 10 );
    anotherStack->push ( 20 );
```

```
    delete aStack;            // Free the object
    delete anotherStack;      // Free the object
}
```

In the first example, the `aStack` and `anotherStack` objects cease to exist when the program leaves the scope of `simplefunction()`, like any automatic variable. However, objects allocated on the heap exist until they are explicitly deleted. C++ provides a `delete` operator that should be used to free the memory used by objects created with `new`.

Encapsulation

One of the most important and useful concepts of object-oriented programming is *encapsulation*. Encapsulation simply means that an object binds some data and the valid operations on that data into a cohesive package. Furthermore, encapsulation implies that the contents of an object cannot be accessed directly by other objects, except through an interface defined by the class.

C++ provides three different levels of encapsulation. Members can be completely accessible to other objects or functions, partially accessible, or completely private to the class. In the previous stack example, the two data members are preceded by the keyword `protected`, which indicates that they are normally not accessible outside the class. For example, if we tried to write a program that contains the statements

```
Stack myStack;
int x = myStack.data[0]; // Error!
```

the C++ translator or compiler would report an error at compile time. Protected data members can only be accessed by the *member functions* that belong to that class, or by member functions that belong to classes *derived* from that class. These new terms will be discussed shortly. C++ also supports a `private` keyword that declares members to be completely private to the class. Only functions declared as part of the class can access private members.

C++ supports a mechanism that allows the programmer to make the internal implementation of a class accessible to other selected functions and classes, without exposing it to the rest of a program. Any class may declare an individual function or an entire class to be a *friend*. Functions declared as friends have complete access to the declaring class's private and protected members. Declaring a class as a friend allows all functions that are members of the friend class to access the declaring class's private and protected members.

Methods and Messages

In common object-oriented terminology, the operations supported by an object are called *methods*. The term *method* comes from the Smalltalk language and has been adopted by many other languages. A method is just a function that is encapsulated within a class, and that can access and operate on the data belonging to objects instantiated from that class. In most object-oriented languages, including C++, there is only one copy of the code that implements each method. The method is defined by the class, but operates on the data of individual instances of that class.

Rather than speaking of "calling" a method as one might call a function, the action of invoking a method is often referred to as "sending" a *message*. Messages are sent to objects, which causes the appropriate class method to be called. The idea of sending messages often provides a very natural way to think about the actions and interactions of objects in a system. For example the objects in the mail system discussed earlier in this tutorial support many messages. In the mail system example, it seems reasonable to talk about sending the `postMaster` object a `deliverMail` message or a `getNewMail` message. This works well because it is easy to visualize the `postMaster` object as a person to whom one can ask, "Do I have any mail?", or "Please deliver my letter First Class."

Some other messages suggested for the mail system don't make as much sense. A mail message seems like an inanimate object, and sending a "print yourself" message doesn't have as clear an analogy in the real world. Still, this type of anthropomorphism is common in object-oriented programming, and is quite useful. It is typical to consider all objects to be active entities, capable of independent action. Thus we can talk about telling a mail message object to "deliver yourself," or a graphical object to "draw yourself."

C++ Member Functions

In C++, methods are called *member function*s. Member functions are simply functions declared as part of a class. In the stack example, `pop()` and `push()`, are member functions that belong to the Stack class. The two other functions, `Stack()` and `~Stack()` are also member functions, but are special. They will be discussed shortly. This tutorial uses the term *method* when discussing object-oriented programming in general, and uses the C++ term, *member function* when discussing C++ classes and functions. When referring to a call to a member function, this tutorial uses either traditional object-oriented terminology ("sending a message to an object") or the terminology favored by C++ ("calling a member function for an object").

Member functions must be declared as part of the class to which they belong. For example, the Stack class declaration shown on page 8 includes a `pop()` member function. Member function implementations must specify the class to which the function belongs by preceding the function name by the class name and two colons. For example, the `pop()` member function can be written like this:

```
int Stack::pop()  // Remove an item from the stack
{
    // Check for underflow before decrementing the
    // stack index and returning a value

    if ( top > 0 )
       return ( data[--top] );
    // Otherwise report an error condition
}
```

Notice that the `pop()` member function can access protected data members of the class, and can also refer to them directly by name, without any qualification. C++ uses a hidden argument to all member functions, whose name is `this`, to implement this feature. The hidden argument provides a pointer to the *instance* on which the function is to operate.[1]

The Stack class's `push()` member function could be written as:

```
void Stack::push ( int item ) // Add an integer to the stack
{
     data[top++] = item;  // Increment the index after adding item
}
```

In C++, the object-oriented notion of sending a message corresponds to invoking the appropriate member function. C++ uses a familiar C-like syntax to invoke member functions, like this:

```
Stack aStack;              // Create a Stack object
aStack.push ( 10 );        // Add an item to the stack
aStack.pop();            // Remove the item
```

If `aStack` is declared as a pointer to a Stack object, member functions are invoked like this:

```
Stack *aStack = new Stack(); // Create a Stack object
aStack->push ( 10 );         // Add an item to the stack
aStack->pop();             // Remove the item
```

Constructors and Destructors

Every C++ class has several special member functions, including a *constructor* and a *destructor*. The constructor is called each time the class is instantiated and provides a way for the programmer to initialize the new object's data members. The constructor always has the same name as the class to which it belongs. So, in the stack example, the `Stack()` member function is the constructor for the Stack class. We can write the Stack constructor as:

```
Stack::Stack()  // Constructor
{
    top  = 0;             // Initialize to next available slot
    data = new int[20];  // Allocate an array of 20 ints
}
```

The Stack constructor initializes the top of the stack to indicate the first entry in the `data` array, and allocates memory for twenty items on the stack.

The class's destructor is called whenever an instance of the class is freed. This can occur because an object declared as an automatic variable goes out of scope, or because a dynamically

---

[1] If the syntax of C++ did not hide this from us, we would see that the equivalent C version of **`pop()`** is written like this:

```
int pop__5StackFv ( register struct Stack *__0this  )
{
   if ( __0this -> top__5Stack > 0 )
      return ( __0this -> data__5Stack [( --__0this -> top__5Stack )] );
}
```

Notice the `__0this` argument that passes a pointer to a Stack structure. Most names are altered to include the name of the class, along with some other information. The process of renaming member functions and data is referred to as *mangling*. Refer to [Stroustrup90] for a concise description of the mangling process. Most C++ implementations support an option that produces the equivalent C code, which is occasionally useful when debugging. It is important to be aware of the hidden **`this`** argument when mixing C and C++.

allocated object is explicitly freed using the `delete` operator. The class destructor should free any memory allocated by the object, and perform any other cleanup that needs to be done when the object is destroyed.

A destructor does not need to explicitly delete the object itself. For example, the Stack destructor only needs to free the memory allocated in the constructor. The destructor has the same name as the class to which it belongs, but is preceded by a tilde ("~") character. We can write the Stack destructor like this:

```
Stack::~Stack()   // Destructor
{
    delete []data;  // Free array allocated by constructor
}
```

Note that C++ requires the array brackets, as shown in this example, when deleting an array. The C++ memory allocator does not maintain information about whether a piece of memory is an array or not. In the situation demonstrated here, no harm would be done by using

```
delete data;
```

However, because C++ would not know that `data` is an array, it would not call destructors for the items in the array. For an integer array like `data`, forgetting the array designation is inconsequential, but this error could pose a problem if `data` was an array of objects.

Inheritance

An essential feature of most object-oriented programming languages is the ability of a class to *inherit* the characteristics of another class. When a class inherits the features of another, the inheriting class is said to be a *subclass* of the other. The class whose features are inherited is known as a *superclass* of the inheriting class. Inheritance relationships can extend over many levels. That is, any given superclass may itself be a subclass of another class.

There are two basic types of inheritance: *single inheritance* and *multiple inheritance*. Figure 4 shows a typical single inheritance hierarchy, in which each class directly inherits the features of one other class, at most.

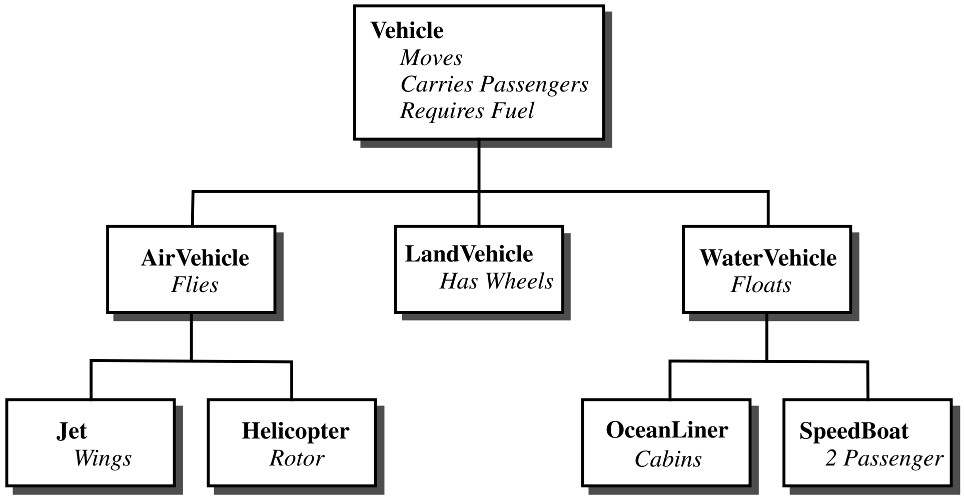

Figure 4 A single inheritance class hierarchy.

Here, the Vehicle class defines some basic characteristics (shown in italics) common to all vehicles. The Vehicle class has three subclasses: AirVehicle, LandVehicle, and WaterVehicle. Each of these classes inherits all the characteristics defined by the Vehicle class, but adds additional features unique to that type of vehicle. In turn, each of these classes has subclasses that inherit the characteristics of Vehicle, plus those of their immediate superclass, and add additional characteristics. So, a Jet is a Vehicle that moves, carries passengers, requires fuel, flies, and has wings.

Multiple inheritance, which is used less frequently, allows a class to inherit the characteristics of more than one immediate superclass.

Inheritance in C++

C++ supports both single and multiple inheritance, although single inheritance is the most common. In C++, a superclass is known as a *base class*, while a subclass is called a *derived class*. As with other object-oriented terminology, this tutorial uses the more traditional object-oriented terms in the general discussion of object-oriented programming in this tutorial and leans toward the C++ terminology in the context of C++ examples.

Let's see how single inheritance works in C++. The stack class described earlier has many deficiencies. First, the stack holds only twenty items. Furthermore, the class does not check for overflow. Inheritance provides a way to extend the Stack class and change or improve it to fit our needs without changing the original class. Let's derive a new class from Stack that checks for stack overflow. This class retains the twenty item limitation.

In C++, a derived class can be declared using the following syntax:

```
class SafeStack : public Stack {

  public:

    SafeStack();        // Constructor
    void push ( int ); // Overrides Stack::push
};
```

This declaration creates a new class, the SafeStack class, which is a subclass of Stack. To use C++ terminology, SafeStack is derived from Stack. The keyword `public` on the first line indicates that the public members of Stack are treated as public members of SafeStack as well. Further, protected members of Stack are treated as protected members of SafeStack. SafeStack does not have access to private members of Stack. The SafeStack class supports the `data` and `top` members, as well as the `pop()` member function provided by the class Stack, just as if SafeStack had declared these members itself.

The SafeStack class redefines the `push()` member function, overriding the `push()` function inherited from the Stack base class. The new member function is written as follows:

```
void SafeStack::push ( int item ) // Add an item to the stack
{
    if ( top < 20 )           // Check for overflow
        data[top++] = item;   // Add item and increment index
    else
       ;   // Handle error condition here
}
```

C++ Constructors, Destructors, and Inheritance

Every class, including derived classes, must have a constructor. If a class declaration does not explicitly include a constructor, C++ creates a default constructor. The SafeStack class requires no initialization and its constructor simply calls its base class's constructor. C++ uses the following syntax for calling base class constructors:

```
SafeStack::SafeStack() : Stack()  // Constructor
{
    // Empty
}
```

This statement arranges for the Stack constructor to be called before executing the body of the SafeStack constructor (which is empty, in this case).

Every class must also have a destructor, although C++ generates a default destructor if the programmer does not provide one. Destructors are called when an object is deleted, in the opposite order in which classes were constructed: destructors that belong to derived classes are executed before those supported by base classes.

Using Inheritance

There are two primary reasons for a programmer to use inheritance. The first is as an implementation convenience. It is often useful to create a new class by inheriting from an existing class that already implements some portion of the features needed in the new class. Often, the new class is a specialization of the original. For example, the SafeStack class discussed earlier added some error checking to the original Stack class. At other times, the new class may have little in common with the chosen superclass beyond the internal implementation. A programmer may choose a superclass because it supports a particularly efficient or complex algorithm. Or, perhaps the superclass simply contains a set of instance variables that are similar to those needed by the subclass.

We can refer to this second kind of inheritance as *implementation inheritance*, because its purpose is to share the implementation of the superclass. Implementation inheritance is useful primarily to the programmer creating the new subclass, because it saves implementation time or provides some other convenience to the programmer.

The other approach is known as *protocol inheritance*. Here, the principal characteristic shared between the subclass and superclass(es) is the external interface, or *protocol*, defined by the superclass. Protocols are discussed further on page 20. Protocol inheritance may or may not benefit the programmer who creates the new subclass, but it almost always benefits the eventual user of the class.

A situation in which protocol inheritance could be used effectively would be a stack class that fixes one or more of the deficiencies of our simple Stack class. For example, suppose we need to write a ResizableStack class that is not limited to twenty members. Even if the ResizableStack class is derived from Stack, we will still need to write a fair amount of code. It might be necessary to change the internal representation of the stack, and the derived class might not even use the two data members supported by the Stack class. All member functions would need to be rewritten.

The class designer would gain no productivity by deriving from the Stack class in this situation. However, the new class would inherit the external protocol of the Stack class, which could be of great importance to the eventual users of the class. This will become clearer as we discuss polymorphism and protocols in the following sections.

The programmer must decide which approach to inheritance is appropriate in any given situation. Happily, these two techniques work well together most of the time. A subclass that inherits the external protocol of another class is likely to share some of its implementation, and vice versa.

Even when a situation involves both protocol inheritance and implementation inheritance, it is important to be very careful when creating inheritance hierarchies. It is easy to misuse inheritance in ways that may be confusing. For example, assume there is an existing OceanLiner class that models a cruise ship. The OceanLiner class probably has characteristics (data members) such as:

cargo	captain	passengerList
engine	fuelTank	schedule

The OceanLiner class would also support operations such as:

```
move        refuel       loadPassengers
```

Now, suppose we need an Airplane class. We *could* derive this class from OceanLiner. An Airplane class would need to support most or all the data members and member functions defined by the OceanLiner class. Inheriting from OceanLiner would save implementation time (it would require less typing), and would also share a large part of the OceanLiner's external protocol. But does it make sense to view an Airplane as a specialization of an OceanLiner? Probably not. This use of inheritance is likely to cause problems later, and will certainly be confusing to anyone who uses the Airplane class.

Situations like this usually indicate the need for some restructuring of the inheritance hierarchy. The Airplane and OceanLiner classes do have something in common, obviously. Each shares the characteristics of a Vehicle, or perhaps a PassengerVehicle. Both should probably be derived from a common ancestor that defines those attributes common to both. But in most cases, they should not inherit from each other.

Polymorphism

An important characteristic of many object-oriented languages is *polymorphism*. Polymorphism allows multiple objects to respond to the same message, while allowing each object to interpret the message in its own way.

A precise (and commonly agreed upon) definition of polymorphism seems to be almost as elusive as a definition of object-oriented programming. The word literally refers to the ability to take on "many forms." Stroustrup [Stroustrup90] calls polymorphism "the ability to call a variety of functions using exactly the same interface - as provided by (C++) virtual functions." Booch [Booch90] leans toward a slightly more theoretical definition and says that "polymorphism is a concept in type theory in which a name may denote objects of many different classes that are related by some common superclass". While Booch's definition may appear at first to have little relationship to Stroustrup's statement, the theoretical concept is the basis of the mechanism that enables polymorphic behavior in C++.

In C++, polymorphism is supported by *virtual* member functions. A function can be declared to be virtual by preceding the function declaration with the keyword `virtual`. For example the following class defines a virtual function, `xyz()`:

```
class X {
  public:
    virtual int xyz();
}
```

Normally, the compiler can determine exactly what member function should be called in any given situation at compile time. However, when a member function is declared to be virtual, such decisions are delayed until runtime. When virtual member functions are used, the actual function to be invoked in response to a given message depends on the dynamic type of the object, not the declared type. To use virtual functions to achieve polymorphic behavior, the objects involved must belong to classes derived from a common base class, and the base class must declare the desired member function to be virtual.

It's easiest to show how virtual functions work with an example. Let's implement two classes, class A and class B. Class A supports one member function, `print()`. For purposes of illustration, this `print()` member function just prints a message reporting the member function and name of the class. Notice that the `print()` member function is *not* declared to be virtual.

```
class A {

  public:

    // A NON-virtual member function
    void print() { cout << "A::print \n"; }
};
```

Now let's derive class B from class A. Class B also supports a similar, non-virtual, `print()` member function:

```
class B: public A {

  public:

    // A NON-virtual member function
    void print() { cout << "B::print \n"; }
};
```

Now, look at the following code that uses classes A and B.The body of the program instantiates two objects, one belonging to class A and the other to class B. The program sends the print message to each object and then passes each object to a function named `printObj()`. This function expects a pointer to an object belonging to class A as an argument. C++ allows pointers to objects that belong to classes derived from the declared type to be passed to this function as well.

```
main()
{
    A *a = new A();     // Create an A object
    B *b = new B();     // Create a B object

    a->print();         // Print the A object
    b->print();         // Print the B object
    printObj ( a );     // Print the A object
    printObj ( b );     // Print the B object
}

void printObj ( A *obj )
{
     obj->print();      // Print the given object
}
```

When this program is executed, it prints the following results:

```
A::print
B::print
A::print
A::print
```

The first three lines seem fine, but what about the fourth line? Because the `printObj()` function expects an object belonging to class A, the `A::print()` member function is executed, even though a class B object is passed to the function. This is probably not what most programmers would intend.

Virtual functions fix this problem. With virtual functions, the member function to be executed depends on the actual (dynamic) type of the object, not the statically declared type. Let's rewrite classes A and B to use a virtual `print()` member function:

```
class A {

  public:

    virtual void print() { cout << "A::print \n"; }
};

class B: public A {

  public:

    void print() { cout << "B::print \n"; } // Virtual because of
                                             // declaration in A
};
```

Now, if we use these classes in the previous example, the results are more like what one would expect:

```
A::print
B::print
A::print
B::print
```

To achieve polymorphic behavior, all classes involved must be derived from a common class. In addition, the common base class must declare all the virtual functions of interest. For example, suppose we create a new class C that adds an additional member function:

```
class C : public A {

  public:

    void print() { cout << "C::print \n"; }
    char *name() { return ( "C" ); }
};
```

Now let's look at an example that includes class C. The following code segment declares three pointers to objects of type A, but instantiates and assigns objects of types A, B, and C:

```
main()
{
    A *a = new A();  // Create an A object
    A *b = new B();  // Create a B object
    A *c = new C();  // Create a C object

    a->print();      // print A
    b->print();      // print B
    c->print();      // print C
    cout << c->name() << "\n"; // Error - A does not support name()
}
```

All three objects can accept and respond appropriately to the `print()` message, but trying to use the `name()` member function causes an error at compile time. Because object "`c`" is declared to belong to class A, the `C::name()` function cannot be accessed. Class A does not support the `name()` member function, and therefore `name()` does not participate in the polymorphic behavior of these classes.

In most cases, the choice of which member functions should be declared as virtual is a design decision, and depends on how the class is being used. However, destructors should almost always be declared virtual. This insures that all class destructors are called correctly when an object is destroyed, regardless of how the object is declared. See [Stroustrup91] for more information on virtual destructors.

Polymorphism can make programs simpler to write and easier to design. The burden of knowing how each type of object implements a particular operation is placed on the class designer instead of the programmer using the class. The programmer who uses a class needs to understand only its external interface and the ultimate result of sending any particular message to an object belonging to that class. Polymorphism also makes systems easier to extend. With polymorphism, existing parts of a system can send the same messages to new objects that support the same methods as other objects already in the system.

Many people maintain that polymorphism is the key to object-oriented programming. Some languages that appear on the surface to be object-oriented support only data abstraction. Data abstraction and encapsulation are very important, but polymorphism adds a new flavor that many feel is an essential part of object-oriented systems.

Programmers that do not use polymorphism often find themselves using `switch` statements to perform different actions depending on the type of a particular piece of data. In object-oriented programming, `switch` statements are viewed with the same disdain previously reserved for the "go to" statement. A `switch` statement in a C++ program should be viewed as a warning that the program may not be structured correctly, and may not be taking advantage of polymorphism.

Protocols

The concept of a *protocol* is closely related to polymorphism. A protocol is simply a well-defined interface, usually made up of the set of methods supported by the class. For example, we could decide that the methods `push()` and `pop()` define the complete protocol required for any stack class. That is, for any class to claim that it obeys the stack protocol, it must support *at least* the `push()` and `pop()` methods. Other stack classes might support additional methods as well, but as long as they support these two methods, we can say they support the stack protocol.

In the example in the previous section, class A defines a protocol in the form of a virtual `print()` member function that is also supported by the B and C derived classes. However, the `name()` member function is not part of the protocol defined by A. Of course, this function could be part of an extended protocol defined for C and its derived classes.

The notion of a protocol is crucial to the design of reusable components and object-oriented architectures in general. Classes with well-defined and enforceable protocols tend to be easier to reuse and maintain. One value of object-oriented programming is that external interfaces tend to be more explicit. Minimizing external references and formalizing the interface between an object and the outside world as much as possible usually results in code that is easier to use and maintain. Such code is usually easier to reuse as well. However, it may be more difficult to write. It is often tempting to throw together something that works and "fix it later," rather than identifying and implementing a complete and correct protocol.

Even when using non-object-oriented techniques, every function or module has an external protocol. However, this external protocol may not be well defined or understood. For example, if a function relies on the value of a global variable, or sets a global variable, then that global variable becomes part of the function's protocol. However, this type of protocol may not be obvious to those who use the function.

Software often exhibits more subtle protocols. For example, the order in which a particular set of functions is called may be important. Such implicit protocols are hard to document, use, and maintain. They provide a likely source of errors because these "hidden" protocols may not be immediately obvious to a programmer who is unfamiliar with a given piece of code. Implicit protocols can be found in object-oriented systems and non-object-oriented systems alike. If a programmer must send a sequence of messages to an object in a particular order, the sequence becomes part of the object's protocol.

When using inheritance, classes have two distinct protocols. The first is the external protocol that determines how the outside world interacts with the class. The second is the protocol between a class and its subclasses. Few object-oriented languages offer any formal way to specify the protocol between a class and its subclasses. However, C++ provides a mechanism that allows the class designer to declare data members and member functions as either private or protected. Protected members cannot be seen by the outside world, but are freely available to derived classes, while private members cannot be seen, even by derived classes. Careful use of the public, private, and protected declarations allow classes to specify both the external and subclass protocols.

Protocols and C++

C++ supports the idea of an *abstract class*, whose primary purpose is to define and enforce a protocol. An abstract class cannot be instantiated. It serves only to specify the external protocol for classes derived from it. In C++, an abstract class is any class that declares a *pure virtual* member function. C++ uses the following syntax to declare a pure virtual function:

```
virtual function() = 0;
```

Any class that declares such a member function cannot be instantiated. All instantiable classes that derive from a C++ abstract class must implement all pure virtual functions. Attempting to instantiate an abstract class directly generates an error at compile time. Pure virtual functions provide a way to enforce a protocol, without implementing all member functions in a base class. The base class simply declares that all instantiable derived classes must implement the function.

For example, the following abstract class defines a stack protocol:

```
class StackProtocol {

   public:

    StackProtocol() { }
    virtual ~StackProtocol() { }
    virtual void push ( int ) = 0;
    virtual int pop() = 0;
};
```

This class does not implement any behavior of its own and contains no data members. It simply declares a set of pure virtual functions that designates a protocol for a stack. Any instantiable class that derives from the StackProtocol class must implement the `push()` and `pop()` member functions, as declared by the StackProtocol class. This forces all derived classes to obey the protocol defined by its base class.

3 Non-object-oriented Features of C++

Although C++ is similar to C in many ways, C++ has many additional features, including some that have little or no direct relationship to object-oriented programming. A few of these features warrant special mention because they enhance the language's ability to support object-oriented programming. The features introduced in the following sections are: inline functions, function overloading, and the use of the `const` declaration.

Inline Functions

Programmers are often concerned with the efficiency of object-oriented systems. For example, object-oriented programs tend to contain many small functions and methods, and the overhead of calling these functions can sometimes become a problem.[1] C++ allows programmers to declare functions as *inline*. When possible, C++ replaces calls to inline functions with actual code in each

[1] Another common concern programmers have about efficiency and object-oriented programming is the cost of the run-time lookup required to implement features like virtual functions. C++ attempts to minimize this cost, and it is seldom a significant factor in real applications.

function, eliminating the cost of a function call. An inline declaration is an optimization request, much like a `register` declaration. The compiler may or may not be able to fulfill the request.

Inline functions are declared with the keyword `inline`, as follows:

```
inline int square ( int x )
{
    return ( x * x );
}
```

The C++ translator or compiler attempts to substitute the body of this function for any call to the function, after making the appropriate parameter substitutions. Note that inline functions are *not* macros, although an inline function can be used effectively in many situations where C programmers would use a macro.

Member functions can also be declared inline, either explicitly or implicitly. Explicit inline member functions are also declared using the `inline` keyword, as in:

```
class MyClass {

  public:

    // Various member functions ...

    inline square ( int );
};

int MyClass::square ( int x )
{
   return ( x * x );
}
```

The body of an inline member function can also be provided inside the class declaration, like this:

```
class MyClass {

  public:

    MyClass();
    int square ( int x ) { return ( x * x ); }
}
```

Here, the `square()` member function is implicitly declared as an inline function and no `inline` keyword is required.

Inline functions are particularly useful for the small access member functions often used in classes. Inline functions allow the programmer to maintain a class's encapsulation, without compromising efficiency.

Notice that the idea of replacing a function call, at compile time, with the code that implements the body of the function is fundamentally at odds with the concept of virtual functions, where the actual function to be called is not known until runtime. Most C++ compilers or translators attempt to deal with inline virtual functions in those cases where the correct function can be determined statically. However, in most cases, inline virtual functions should be avoided.

Function Overloading

C++ allows the programmer to declare multiple functions with the same name, as long as the functions require different argument types or different numbers of arguments. Such functions are said to be *overloaded*. Overloaded functions can be very useful in writing C++ classes.

For example, assume that the mail system example discussed earlier also supports several possible types of output devices (printers), and that these devices are modeled as objects. We could have a LinePrinter class, a LaserPrinter class, and a Typesetter class. We could now define the print protocol for all objects in the mail system in a PrintableObject class that supports these different printers.

The PrintableObject class might need to prepare or format the data differently for different types of printers. We could handle this situation by designing the class to support member functions like `printToLaser()`, `printToLinePrinter()`, and so on, but the result would be very awkward. Another approach would be to pass an argument to the print function specifying the desired type of printer. A single function could check the printer object's type and perform the action appropriate for the type.

Overloaded functions allow programmers to achieve the same effect without checking the type at runtime. For example, C++ allows the multiple printer scenario to be handled as follows:

```
class PrintableObject {

  public:

    // Other member functions

    print ( LinePrinter * );
    print ( LaserPrinter * );
    print ( Typesetter * );
    print ( LinePrinter *, LaserPrinter * ); // Print to both
}
```

Now, the PrintableObject class supports four different member functions, all named `print()`. Which member function is called in response to any given `print` message depends on the number and type of the arguments. Notice that unlike virtual functions, the compiler can determine which overloaded function is called at compile-time. Member functions can, of course, be both virtual and overloaded at the same time.

It is often convenient to overload a class constructor to provide more than one way to initialize a class. For example, we might wish to implement a ResizableStack class that supports either a variable or a fixed sized stack. This class could have two constructors, declared as:

```
class ResizableStack {

  public:

    // Various members

    ResizableStack();               // Use a dynamically sized stack
    ResizableStack ( int size );    // Pre-allocate initial stack size
    // ...
```

The constructor with no arguments would allocate a small stack, using some default size. Applications that anticipate storing a specific amount of data can call the other constructor to provide this information to the class.

C++ also allows function prototypes to declare default values for some parameters. For example, the above example could also be implemented with a single constructor, with an optional argument. This is done by assigning a default value to the parameter in the prototype, as shown in the following example:

```
class ResizableStack {

  public:

    // Various members

    ResizableStack ( int size = 20 ); // Stack size is 20 by default
    // ...
```

The const Declaration

C++ allows the programmer to declare variables, as well as values passed or returned from functions, to be immutable, or `const`. A variable declared to be `const` cannot be altered. The `const` declaration is particularly useful when writing object-oriented programs in C++ because it allows the programmer to improve the encapsulation of a class when passing or returning pointers.

The `const` declaration is particularly useful for improving encapsulation in C++ classes. For example, the following String class supports an access function that returns a character string. Because a string is just an array of characters, simply returning the address of the string would allow the string to be manipulated, and even overwritten, from outside the class. The following implementation uses the `const` declaration to prevent the string from being modified outside the class.

```
class String {

  private:

    char *_data;       // The characters in the string
    int   _length;     // Number of characters in string

  public:
```

```
    String ( char *str ) // Constructor - declared inline
    {
        _data = str; _length = strlen ( str );
    }
    const char *const data() { return ( _data ); } // Accessor for _data
};
```

Here, the `data()` function is declared to return a const pointer to a const value. Neither the pointer to the string, nor the string itself can be modified. This use of `const` does impose some restrictions on how the class can be used. For example, we cannot assign the result of the `data()` member function to another non-`const` variable:

```
String *nameString = new String ( "George" );
char   *name;
name = nameString->data();                // Error!
if ( strcmp ( name, "George" ) != 0 )
    // ...
```

However, because `data()` is an inline function, it can usually be used directly wherever it is needed, with no loss of efficiency, and without compromising the encapsulation of the String class:

```
String *nameString = new String ( "George" );

if ( strcmp ( nameString->data(), "George" ) != 0 )
    // ...
```

4 Summary

This tutorial introduced some basic object-oriented concepts and discussed the mechanisms that support object-oriented programming in C++. The syntax of C++ is very similar to that of C, and it is possible to program in C++ using the same style used to program in C. However, to get the most out of C++, programmers should learn to use the new facilities provided by the language, and develop an object-oriented programming style. In addition to classes, member functions, and inheritance, C++ also supports many useful non-object-oriented features such as inline functions and function overloading.