Understanding C++: An Accelerated Introduction

by Marshall Brain

Introduction

For many people the transition from C to C++ is not easy. In fact, this transition is often accompanied by quite a bit of anxiety because C++ is surrounded by a certain aura that makes it inaccessible. For example, you can pick up a book on C++, randomly turn to a page, and encounter paragraphs like this:

"From a design perspective, private derivation is equivalent to containment except for the (occasionally important) issue of overriding. An important use of this is the technique of deriving a class publicly from an abstract base class defining an interface and privately from a concrete class providing an implementation. Because the inheritance implied in private derivation is an implementation detail that is not reflected in the type of the derived class, it is sometimes called 'implementation inheritance' and contrasted to public declaration, where the interface of the base class is inherited and the implicit conversion to the base type is allowed. The latter is sometimes referred to as sub- typing or 'interface inheritance'." [From "The C++ Programming Language, second edition", by Bjarne Stroustrup, page 413]

It can be difficult to get started in an environment that is this obtuse.

The goal of these tutorials is to help you to gain an understanding of the fundamental concepts driving C++ in a quick and painless way. They let you begin thinking in an "object oriented way". Once you understand the fundamentals, the rest of the language is relatively straightforward because you will have a framework on which to attach other details as you need them. Once you understand its underlying themes and vocabulary, C++ turns out to be a remarkable language with quite a bit of expressive power. Used correctly, it can dramatically improve your productivity as a programmer.

These tutorials answer three common questions:

1. Why does C++ exist, and what are its advantages over C?
2. What tools are available in C++ to express object oriented ideas?
3. How do you design and implement code using object oriented principles?

Once you understand the basic tools available in C++ and know how and why to use them, you have become a C++ programmer. These tutorials will start you down that road, and make other C++ material (even Stroustrup) much easier to understand.

These tutorials assume that you already know C. If that isn't the case, spend a week or two acclimating yourself to the C language and then come back to these tutorials. C++ is a superset of C, so almost everything that you know about C will map straight into this new language.

1.1 Why does C++ Exist?

People who are new to C++, or who are trying to learn about it from books, often have two major questions: 1) "Everything I read always has this crazy vocabulary--'encapsulation', 'inheritance', 'virtual functions', 'classes', 'overloading', 'friends'-- Where is all of this stuff coming from?" and

2) "This language--and object oriented programming in general--obviously involve a major mental shift, so how do I learn to think in a C++ way?" Both of these questions can be answered, and the design of C++ as a whole is much easier to swallow, if you know what the designers of C++ were trying to accomplish when they created the language. If you understand why the designers made the choices they did, and why they designed certain features into the language, then it is much easier to understand the language itself.

Language design is an evolutionary process. A new language is often created by looking at the lessons learned from past languages, or by trying to add newly conceived features to a language. Languages also evolve to solve specific problems. For example, Ada was designed primarily to solve a vexing problem faced by the Pentagon. Programmers writing code for different military systems were using hundreds of different languages, and it was impossible to later maintain or upgrade the systems because of this. Ada tries to solve some of these problems by combining the good features of many different languages into a single language.

Another good example of the evolutionary process in computer languages occurred with the development of structured languages. These languages arose in response to a major problem unforeseen by earlier language designers: the overuse of the goto statement in large programs. In a small program, goto statements are not a problem. But in a large program, especially when used by someone who *likes* goto statements, they are terrible. They make the code completely incomprehensible to anyone who is trying to read it for the first time. Languages evolved to solve this problem, eliminating the goto statement entirely and making it easier to break large programs down into manageable functions and modules.

C++ is an "object oriented" language. Object oriented programming is a reaction to programming problems that were first seen in large programs being developed in the 70s. All object oriented languages try to accomplish three things as a way of thwarting the problems inherent in large projects:

1. Object oriented languages all implement "data abstraction" in a clean way using a concept called "classes". We will look at data abstraction in much more detail later because it is a central concept in C++. Briefly, data abstraction is a way of combining data with the functions used to manipulate the data so that implementation details are hidden from the programmer. Data abstraction makes programs much easier to maintain and upgrade.

2. All object oriented languages try to make parts of programs easily reusable and extensible. This is where the word "object" comes from. Programs are broken down into reusable objects. These objects can then be grouped together in different ways to form new programs. Existing objects can also be extended. By giving programmers a very clean way to reuse code, and by virtually forcing programmers to write code this way, it is much easier to write new programs by assembling existing pieces.

3. Object oriented languages try to make existing code easily modifiable without actually changing the code. This is a unique and very powerful concept, because it does not at first seem possible to change something without changing it. Using two new concepts however --*inheritance* and *polymorphism*-- it is possible to do just that. The existing object stays the same, and any changes are layered on top of it. The programmer's ability to maintain and adjust code in a bug-free way is drastically improved using this approach.

Since C++ is an object oriented language, it possesses the three object oriented benefits discussed above. C++ adds two other enhancements of its own to clean up problems in the original C language or to make programming in C++ easier than it is in C:

1. C++ adds a concept called "operator overloading". This feature lets you specify new ways of using standard operators like "+" and "> > " in your own programs. For example, if you want to add a new type such as a complex number type to a C program, the implementation will not be clean. To add two complex numbers, you will have to create a function named "add" and then say "c3=add(c1,c2);", where c1, c2 and

c3 are values of the new complex number type. In C++, you can *overload* the "+" and "=" operators instead, so that you can say, "c3 = c1 + c2". In this way, new types are added to the language in a completely seamless manner. The overloading concept extends to all functions created in C++.

2. C++ also cleans up the implementation of several portions of the C language, most importantly I/O and memory allocation. The new implementations have been created with an eye toward operator overloading, so that it is easy to add new types and provide seamless I/O operations and memory allocation for them.

Let's look at some examples of problems that you have probably run across in your C programming exploits, and then look at how they are solved in C++.

The first example can be seen in every library that is built in C. The problem is demonstrated in the code below, which sets a string to a value and then concatenates another string onto it:

```
char s[100];
strcpy(s, "hello ");
strcat(s, "world");
```

This code is not very pretty, but the format is typical of every library you create in C. The string type is built out of the array-of-characters type native to C. Because the new type is not part of the original language, the programmer is forced to use function calls to do anything with it. What you would like to do instead is be able to create a new type and have it seamlessly blend in with the rest of the language. Something like this:

```
string s;

s = "hello ";
s += "world";
```

If this were possible, then the language would be infinitely extensible. C++ supports this sort of extension through operator overloading and classes. Notice also that by using the **string** type, the implementation is completely hidden. That is, you do not know that--or if--**string** has been created using an array of characters, a linked list, etc., and it appears to have no maximum length. Therefore it is easy to change the implementation of the type in the future without adversely affecting existing code.

Another example using a library can be seen in the implementation of a simple stack library. The function prototypes for a typical stack library (normally found in the header file) are shown below:

```
void stack_init(stack s, int max_size);
int stack_push(stack s, int value);
int stack_pop(stack s, int *value);
void stack_clear(stack s);
void stack_destroy(stack s);
```

The user of this library can push, pop and clear the stack, but before these operations are valid the stack must be initialized with **stack_init**. When finished with the stack, the stack must be destroyed with **stack_destroy**. But what if you forget the initialization or destruction steps? In the former case, the code will not work and it can be very difficult to track down the problem unless all of the routines in the library detect initialization failure and report it. In the latter case, the failure to destroy the stack properly can cause memory leaks that are again very difficult to track down. C++ solves this problem using *constructors* and *destructors*, which automatically handle initialization and destruction of objects such as stacks.

Continuing with the stack example, notice that the stack as defined can push and pop integers. What if you want to create another stack that can handle reals, and another for characters? You will have to create three separate libraries, or alternatively use a union and let the union handle all different types possible. In C++, a concept called templates lets you create just one stack library and redefine the types stored on the stack when it is declared.

Another problem that you might have had as a C programmer involves changing libraries. Say, for example, that you are using the **printf** function defined in the stdio library but you want to modify it so that it can handle a new type you have recently created. For example, you might want to modify **printf** so that it can print complex numbers. You are out of luck unless you happen to have the source code for **printf**. And even if you have the source, modification won't do a lot of good because that source is not portable, nor do you have the right to copy it. There really is no way to extend a C library easily once it has been compiled. To solve your output problem, you will have to create a new function to print your new type. If you have more than one new type, then you probably will have to create several different output functions, and they will all be different. C++ handles all of these problems with its new technique for standard output. A combination of operator overloading and classes allow new types to integrate themselves into the standard C++ I/O scheme.

While thinking about the **printf** function, think about its design and ask yourself this: Is that a good way to design code? Inside of **printf** there is a **switch** statement or an if-else-if chain that is parsing the format string. A %d is used for decimal numbers, a %c is used for characters, a %s is used for strings, and so on. There are at least three problems with this implementation:

1. The programmer has to maintain that switch statement and modify it for each new type that is to be handled. Modification means that new bugs might be introduced.

2. There is no guarantee that the user will match up the data parameters with the format string, so the whole system can fail catastrophically.

3. It is inextensible--unless you have the source you cannot extend the **printf** statement.

C++ solves these problems completely by forcing the programmer to structure the code in a new way. The switch statement is hidden and handled automatically by the compiler through *function overloading*. It is impossible to mismatch the parameters, first because they are not implemented as parameters in C++, and second because the type of the variable automatically controls the switching mechanism that is implemented by the compiler.

C++ solves many other problems as well. For example, it solves the "common code replicated in many places" problem by letting you factor out common code in a third dimension. It solves the "I want to change the parameter type passed into a function without changing the function" problem by letting you overload the same function name with multiple parameter lists. It solves the "I want to make a tiny change to the way this works, but I don't have the source for it" problem, and at the same time it also solves the "I want to redo this function completely but not change the rest of the library" problem using inheritance. It makes the creation of libraries much cleaner. It drastically improves the maintainability of code. And so on.

You have to change your way of thinking slightly in order to take advantage of much of this power, and it turns out that you generally have to consider the design of your code up front a little more. If you don't, you lose many of the benefits. As you can see however, you gain a great deal in return for your investment. As in everything else, there is a tradeoff, but overall the benefits outweigh the disadvantages.

Understanding C++: An Accelerated Introduction

by Marshall Brain

Improving C

Everything you have ever written in C works in C++. However, in many cases C++ offers a better way to handle a given task. In other cases C++ offers a second way to do something, and the option gives you more flexibility. In this section we will examine C++ extensions to C. Many of these extensions were not added for their own sake, but instead "enable" object oriented features that we will see in later tutorials.

This tutorial contains a lot of detail. Don't panic--just scan it for now if you like, and then come back and study the necessary sections as they are needed later on. These concepts have been collected here for easy reference because they are used at many different places in the notes.

2.1 Comments

C++ supports the old-style multi-line C comment, as well as a new single line form denoted by the "//" symbol. For example:

```
// get_it function reads in input values
void get_it()
{
    // do something.
}
```

Everything from the "//" to the end of the line is ignored. You can use both commenting styles interchangeably in a C++ program.

2.2 Type casting

In C, you cast a type by placing a type name in parenthesis and placing it in front of the variable name, as shown below

```
int i;
float f;

f = (float) i;
```

In C++ a second format is also supported. It makes the cast look like a function call, as shown here

```
int i;
float f;

f = float(i);
```

We will see later, when we begin discussing classes, that there is a reason for this new format.

2.3 Input and output

2.3.1 Terminal I/O

One of the most obvious differences between C and C++ is the replacement of the stdio library in C with the iostream library in C++. The iostream library takes advantage of a number of the features of the C++ object-oriented extensions (we will see detailed examples later), and therefore makes the addition of new user-defined type I/O much easier. The iostream library also replaces all of the capabilities found in the stdio library, so it is important to know how to use the basic features of the new library as you translate code over to C++.

Use of the iostream library for basic input and output is straightforward. Two simple examples are shown below:

```
cout <<  "hello\n";
```

or equivalently:

```
cout <<  "hello" <<  endl;
```

Both forms produce the same output, and cause the word "hello" followed by a newline to appear on standard out. The word **cout** indicates stdout as the destination for the output, and the **<<** operator (the *insertion operator*) is used to gather the items. Two other standard output destinations are pre-defined: **cerr** for unbuffered error information, and **clog** for buffered error information.

Any of the standard types can be written using the technique shown above: integers, floats, characters, and pointers to characters all can be written. Multiple items can either be strung together on a single line or stacked on multiple lines. For example:

```
int i = 2;
float f = 3.14
char c = 'A';
char *s = "hello";

cout << s << c << f << i <<  endl;
```

produces the output:

```
helloA3.142
```

and it is the same as:

```
cout << s << c;
cout << f;
cout i << endl;
```

The **cout** mechanism automatically understands addresses, and formats them for hex output. For example, if **i** is an integer then the statement:

```
cout << &i << endl;
```

prints the address of **i** in hex format. If **p** is a pointer to **i**, then printing **p** also prints the address of **i** in hex format. There are cases however where this formatting rule does not hold. Printing **s**, where **s** is a pointer to a character, produces the string pointed to by **s** rather than the address held by **s**. To remedy this situation, cast **s** to a void pointer as shown below if you want to see its address:

```
cout << (void *) s;
```

Now the address held by **s** will be shown in hex format. If you wish to display an address as a decimal number rather than in hex format, cast it to a **long** integer:

```
cout << long(& i);
```

This line prints the address of **i** in decimal format. In the same way, an **int** cast is used to print out the integer value of a character:

```
cout << int('A');        // produces 65 as output
```

You may notice that the << operator--known in C as the shift left operator--has been stolen to handle output in C++. If you wish to use it for shifting left within an output line, then parenthesis should be used:

```
cout << (2 << 4);        // produces 32 as output
```

To format output, you can use several techniques. Information can be spaced by adding in spaces or tabs as literal strings, as shown below:

```
int i = 2;
float f = 3.14
char c = 'A';
char *s = "hello";

cout << s << " " << c << "\t" << f
    << "\t" << i << endl;
```

There are several other manipulators that can be inserted into an output stream (on many systems you will have to include "iomanip.h" to use these)

- dec Use decimal base
- oct Use octal base
- hex Use hex base
- endl End of line
- ends End of string ('\0')
- flush Flush output buffer
-
- setw(w) Set output width to w (0 is default)
- setfill(c) Set fill character to c (blank is default)
- setprecision(p) Set float precision to p

The statement:

```
cout << "[" << setw (6) << setfill('*') << 192;
cout << "]" << endl;
cout << hex << "[" << setw (6);
cout << setfill('*') << 192 << "]" << endl;
cout << setprecision(4) << 3.14159 << endl;
```

produces:

```
[***192]
[****c0]
3.142
```

Floating point output may or may not truncate trailing zeros no matter how you set the precision--it is compiler-dependent.

You can see from the above examples that certain variable and function names should not be used to avoid losing the manipulators built into the iostream library.

Input is handled in a similar manner, using the **cin** input stream and the ">>" *extraction* operator. For example, the statement:

```
int i,j,k;
cin >> i >> j >> k;
```

will read three integer values from stdin into **i**, **j** and **k**. White space is automatically used as a separator and ignored. When reading into a string variable, the input is read word by word, where words are separated by white space. White space characters are ignored when reading into a character. This behavior can be overridden by explicitly reading strings and lines (see below). All of the standard types handled by **cout** are handled by **cin**. The **cin** stream can also be used in a while loop that terminates when EOF is detected, as shown below:

```
while (cin >> i)
    cout <<  i;
```

The **cin** stream automatically breaks string input into words and terminates on EOF.

2.3.2 File input and output

Input and output to text files are handled by including the file "fstream.h" and by then declaring variables of type **ifstream** and **ofstream** respectively. For example, the following program reads from a file named "xxx" and writes to a file named "yyy":

```
#include <iostream.h>
#include <fstream.h>

void main()
{
    char c;
    ifstream infile("xxx");
    ofstream outfile("yyy");
    if (outfile &&  infile) // They will be 0 on err.
        while (infile >> c)
            outfile <<  c;
}
```

The **infile** and **outfile** variables are passed the file name on initialization, and are used just as **cin** and **cout** are used. This code does not do work as expected however, because blanks, tabs, and '\0' characters at the end of each line are ignored as white space when using << on a character. Instead, the "get" function can be used, as shown below:

```
while (infile.get(c))
    outfile << c;
```

or:

```
while (infile.get(c))
    outfile.put(c);
```

It is also possible to read complete lines by calling the "getline" function in the same manner as used for the "get" function. To open a file for appending, use the following:

```
ofstream("xxx", ios::app);
```

This line, along with the ".get" function notation, will make more sense once you know more about C++. The fact that **ofsteam** sometimes takes one parameter and other times takes two is built into C++ (see Section 2.6).

Note that no "close" function is needed for file input and output. A file automatically closes itself when the file variable goes out of scope. If you do need to explicitly close a file, you can say:

```
outfile.close();
```

2.3.3 String I/O

Input can be read from strings in memory, and output can be sent to strings in memory, duplicating the action of **sscanf** and **sprintf**. To do this, you must include the file "strstream.h" and then declare input and output strings. An output string is shown below:

```
char s[100];
ostrstream outstring(s,100);

outstring << 3.14 << " is pi" << ends;
cout << s;
```

The string **s** is filled with the text "3.14 is pi". If **s** is overfilled, **outstring** will automatically stop placing values into it.

If a string s exists and you wish to read from it, you can use an input string stream as shown below:

```
char *s = "3.14  12  cat";
```

```
istrstream instring(s, strlen(s));
float f;
int i;
char t[100];

instring >> f >> i >> t;
```

The iostream library has many many other capabilities not discussed here. For more information see the the C++ documentation supplied with the compiler--it contains a complete reference on the I/O library.

2.4 Variable declarations

Variables are declared in C++ as they are in C. Variables can be declared anywhere in the code in C++, returning things almost to the point of FORTRAN in terms of flexibility. The variable comes into existence when it is declared, and ceases to exist when the ending brace of the current block of code is reached. For example, in the following code:

```
{
    int i;
    ... code ...
    int j;
    ... code ...
    int k=func(i,j);
    ... code ...
}
```

All three variable come into existence at the point of declaration and disappear at the closing brace.

2.5 Constants

In C you create a constant by using the macro preprocessor. An example is shown below:

```
#define MAX 100
```

When the program is compiled, the preprocessor finds each occurrence of the word MAX and replaces it with the string 100.

In C++, the word "const" is used instead, and it is applied to normal variable declarations as shown below:

```
const int MAX=100;
```

The **int MAX=100;** portion is formatted exactly the same way as a normal declaration. The word **const** in front of it simply defines that the variable MAX cannot be subsequently changed.

The use of uppercase characters for constant variable names is a C tradition which you may choose to uphold or ignore.

The **const** modifier can also be used in parameter lists to specify the valid usage of a parameter. The three functions below demonstrate different uses of const.

```
void func1(const int i)
{
    i=5;        // cannot modify a constant
}

void func2(char * const s)
{
    s="hello";  // cannot modify the pointer
}

void func3(const char * s)
{
    s="hello";  // this is OK
    *s='A';     // cannot modify what is pointed to
}
```

The usage shown in **func2** should almost always be used when a **char*** parameter is passed.

2.6 Function overloading

One of the most powerful new features in C++ is called "function overloading". An overloaded function has several different parameter lists. The language distinguishes which function to call based on pattern- matching the parameter list types. Here is an extremely simple demonstration of the process:

```
#include <iostream.h>

void func(int i)
{
    cout << "function 1 called" << endl;
    cout << "parameter = " << i << endl;
}

void func(char c)
{
    cout << "function 2 called" << endl;
    cout << "parameter = " << c << endl;
}

void func(char *s)
{
    cout << "function 3 called" << endl;
    cout << "parameter = " << s << endl;
}

void func(char *s, int i)
{
    cout << "function 4 called" << endl;
    cout << "parameter = " << s;
    cout << ", parameter = " << i << endl;
}

main()
{
    func(10);
    func('B');
    func("hello");
    func("string", 4);
    return 0;
}
```

When this code is executed, each version of the function **func** is called based on parameter list matching. You will use this capability a great deal in C++ once you get used to the idea. For example, if you create a function that initializes a module, you can have it call different code depending on whether it is passed a string, an integer, a float, and so on.

2.7 Default arguments

C++ also allows you to give default values to parameters--if the parameter is not passed, the default value is used. This capability is demonstrated in the following code:

```
#include <iostream.h>

void sample(char *s, int i=5)
{
    cout << "parameter 1 = " << s << endl;
    cout << "parameter 2 = " << i << endl;
}

main()
{
    sample("test1");
    sample("test1",10);
    return 0;
}
```

The first function call will output the default value of 5 for the parameter **i**, while the second call will output the value 10.

When creating default parameters, you need to avoid ambiguity between the default parameter lists and other overloaded parameter lists. For example, given the above function definition for **sample** it is not possible to create an overloaded version that accepts a single **char*** parameter--the compiler would be unable to pick which function to call in the case where it is passed a string.

2.8 Memory allocation

C++ replaces the C memory allocation function **malloc** and the deallocation function **free** with **new** and **delete** respectively, and in the process makes them much easier to use. **New** and **delete** allow user-created types to be allocated as easily as existing types.

The following code shows the simplest use of new and delete. A pointer to an integer is points to a block of memory created by **new**:

```
int *p;
p = new int;
*p = 12;
cout << *p;
delete p;
```

It is also possible to allocate blocks consisting of arrays of varying length using a similar technique. Note the use of **delete []** for deleting the array:

```
int *p;
p = new int[100];
p[10] = 12;
cout << p[10];
delete [] p;
```

The value 100 can be a variable if desired.

When working with user-defined types, **new** works just the same way. For example:

```
typedef node
{
    int data;
    node *next;
} node;

main()
{
    node *p;
    p=new node;
    p->data = 10;
    delete p;
}
```

We will see in later tutorials that the **delete** operator is very sophisticated when working with user-defined classes.

2.9 Reference declarations

In C, pointers are frequently used to pass parameters to functions. For example, the following **swap** function swaps the two values passed:

```
void swap(int *i, int *j)
{
    int t = *i;
    *i = *j;
    *j = t;
}

main()
{
    int a=10, b=5;

    swap(& a, & b);
    cout << a << b << endl;
}
```

C++ provides a *referencing* operator to clean up the syntax a bit. The following code works in C++:

```
void swap(int&  i, int&  j)
{
    int t = i;
    i = j;
    j = t;
}

main()
{
    int a=10, b=5;

    swap(a, b);
    cout << a << b << endl;
}
```

The parameters **i** and **j** declared as type **int&** act as references to the integers passed (read **int&** as "a reference to an integer") . When a variable is assigned to the reference variable, the reference picks up its address and mimics the actual location of the assigned variable. For example:

```
int a;
int & b=a;

a=0;
b=5;
cout << a << endl;
```

This code produces 5 as its output because **b** references **a**. It is the same as using pointers and address operators in C but the syntax has been greatly simplified. Note that **b** must be initialized at creation as shown.

Understanding C++: An Accelerated Introduction

by Marshall Brain

C++ Vocabulary

The last tutorial focused on elements of the C++ language that extend C or correct problems inherent in it. These modifications are fairly easy to understand. The other part of C++ is the object oriented extensions. These additions to the language are not so easy to understand. Whereas the **cout** capability is simply another way to handling printing--which you already understand--many of the object oriented extensions will be unfamiliar. The purpose of this chapter is to give you your first exposure to some of the general ideas. Then we will look at the C++ syntax that supports these concepts and come back and look at the concepts again.

3.1 C++ Vocabulary

Look at the world around you. You can understand a good bit about the structure, vocabulary, and organization of C++ by looking at the structure and organization of the real world as well as the vocabulary that we use to talk about it. Many of the design elements of C++--and object oriented languages in general--try to emulate the way we interact with the real world.

For example, whenever you look around yourself you see a large number of objects. We organize all of the objects around us in our minds by arranging them in hierarchical categories, or "classes". For example, you have in your hands a book. A book is a general class of object. You might say, "This object I am holding is classified as a book."

A hierarchy of object classes surrounds the class "book", and it extends in two directions. Books are a member of the more general class "publications". Specific types of books also exists: computer books, fiction books, biographical books, and so on. The hierarchy extends both toward the general and the more specific. At this point you are holding a single, particular book. In OOP lingo, you are holding an "instance" of the class "book".

Books have certain attributes that are shared by all books: They have a cover, several chapters, no advertising, and so on. They also have attributes shared by publications in general: a title, a date of publication, a publisher, etc. They have attributes that are shared by all physical objects: a location, size, shape, and weight. This idea of shared attributes is very important in C++. C++ models the concept of shared attributes using *inheritance*.

There are certain things you do with and to different objects, and those actions change from object to object. For example, you can read a book, and you can flip its pages. You can look at the title, find a specific chapter, look something up in the index, count the number of pages, and so on. These actions are largely unique to publications: you never find yourself flipping the pages of a hammer, for example. However, there are actions that are generic to all physical objects, such as picking them up. C++ takes this fact about the world into account as well, again using inheritance.

The hierarchical nature of object categories, as well as our hierarchical organization of object attributes and actions, are all embedded into the syntax and vocabulary of C++. For example, when designing a program you will break it down into objects, each of which has a "class". You will "inherit" features of a "base class" when you create a "derived class". That is, you will create general object classes and then make more specific classes from them, deriving the particular from the general. You will "encapsulate" the data found in an object with "member functions", and as you extend a class you will "overload" and "override" the functions of the base class. Confused? Let's look at a quick example to see what all of these words actually mean.

The classic example of object oriented programming is a graphics program that allows you to draw objects--lines, rectangles, circles and such--on the screen. What do all of these objects share in common? All objects have a location on the screen. They might also have a color. These attributes are possessed by every shape shown on the screen. Therefore, as a program designer you would create a "base class"--another way to think about it is "a generic object class"--that holds attributes found in all objects appearing on the screen. The base class might be called "shape" to identify it in a generic sort of way. You would then "derive" different objects--circles, squares, lines--from this base class, adding in new attributes that are specific to these objects. A specific circle drawn on the screen using the circle class would then be an "instance" of the circle class, which inherited some of its behavior from the more generic shape class.

It is possible to create this sort of hierarchy with normal structures in C, but it is not nearly as easy to do as it is in C++. C++ contains syntax to handle inheritance. For example, in C you could create a base structure that holds the object's location on the screen and color. Then specific object structures could include this base structure and add to it. C++ makes this process easier, and then goes one step further. In C++, functions can be bonded into a structure as well and this is called a "class". So the base class might have "member functions", as they are called in C++, that allow an object to be moved and recolored. The "derived classes" can use these member functions as they are, or add in new member functions to increase functionality, or override existing member functions to change behavior.

The most important feature differentiating C++ from C is this idea of a "class", both at a syntactic and a conceptual level. Classes let you use all of the normal object oriented programming features--encapsulation, inheritance, and polymorphism--in your C++ programs. They also are the framework on which other features, such as "operator overloading" (the ability to redefine operators such as "+" and ">" for newly created data types), are built. That all may sound like gibberish now, but as you become familiar with the concepts and vocabulary you will begin to see the power of these new techniques.

3.2 The evolution of classes

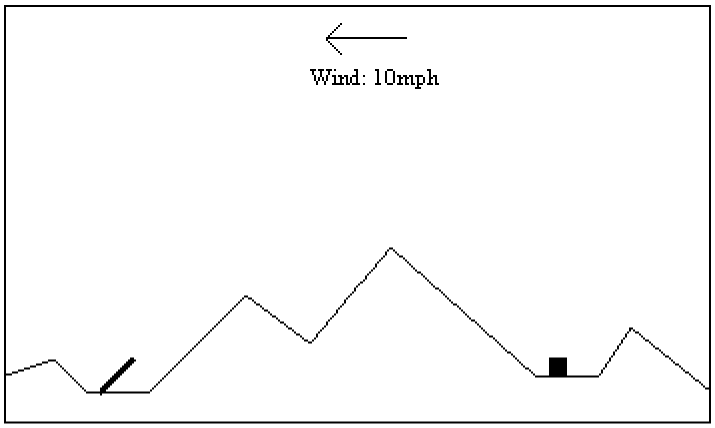
Given the amount of conceptual power embodied in the class concept, it is interesting to note that the syntax remains fairly straightforward. A class is simply an extension of a C structure. Basically a class allows you to create a structure, and then permanently bind all related functions to that structure. This process is known as *encapsulation*. It is a very simple concept, but it is the heart of object oriented programming: *data + functions = object*. Classes can also be built on top of other classes using *inheritance*. Under inheritance, a new class extends its base class. Finally, new classes can modify the behavior of their base classes, a capability known as *polymorphism*.

This is a new way of thinking about your code--it is three-dimensional thinking. You can consider a straight-line piece of code (one that has no functions) as one-dimensional code. It starts at the beginning and ends at the end and that's it. Then you add functions to it to remove some of the redundancies and give names to some of the big pieces. That's two-dimensional code. Now we are going to add a third dimension to that, grouping functions and data together into classes so that the code is further organized. The class hierarchy created by inheritance adds the third dimension. And just as flying is much harder to master than driving because flying adds a third dimension to the mix, object-oriented programming can take some time to master.

One of the best ways to understand classes and their importance to you as a programmer is to understand how and why they evolved. The roots of the class concept lie in a topic known as "data abstraction".

Let's imagine that you are watching a typical room full of college freshmen write a program. Imagine a group of such students who are in their first-semester Pascal course. Once they know how to create **if** statements and loops and arrays they are pretty much ready to write code, but they don't yet know how to organize their thinking. If you ask them to create a simple program they create a blob of code that does the job somehow. It won't be pretty, but it will work.

Imagine that you have asked these students to create a program that can play the "cannon" game. If you have been around computers for 15 years or so then you are familiar with this game because it was very common on early personal computers: The player sees a cannon and a target sitting on terrain that changes from game to game. The goal is to set the angle of the cannon and the amount of powder so that the cannon ball hits the target, missing any hills or other obstacles in the terrain.



Assume that the terrain data exists in a text file consisting of pairs of coordinates. The coordinates are endpoints of the line segments that define the terrain. The students figure out that they need to read this file in so that they can draw it, and they also need it in memory so that they can check for intersections of the cannon ball's path with the terrain in order to determine where the cannon ball "lands". So what do they do? They declare a global array to hold the coordinates, read the file into the array, and then use the array whenever it is needed anywhere in the program.

The problem with this approach is that the array has now embedded itself in their code. If a change is ever required--say from an array to a linked list--the program will probably be thrown out and rewritten because it contains so many specific references to the array that change is impossible. From a production-programming standpoint this is not good because data structures frequently change in a large program.

A better way to design the program is to use an "abstract data type". In this approach, the programmer first tries to decide how the data will be used. In our terrain example, the programmer might think, "Well, I need to be able to load in the terrain data from wherever it comes from, and to draw the terrain on the screen, and to see if the cannon ball's path intersects with the terrain." Notice that this is done abstractly--there is no mention of an array or a linked list anywhere. Then the programmer creates functions to implement those capabilities. The functions might be named **load_terrain**, **draw_terrain**, and **check_terrain_intersection**. These functions are used throughout the program.

The functions act as a barrier. They hide the actual data structure from the program. If the data structure later has to change, say from an array to a linked list, the majority of the program remains unaffected--only the three functions have to change. The programmer has succeeded in creating an "abstract data type".

Many languages formalize this concept. In Pascal you can use a "unit", in C you can use a "library". Both allow you to create and separately compile a file containing the data structure and the functions that access it. You can specify that the data structure be "hidden", which means that the array can only be accessed by the functions in that unit. In addition, the unit can be compiled so that the code inside is hidden as well: other programmers can call the functions because of a publicly available interface, but they cannot see or modify the actual code.

Pascal units and C libraries represent a step in an evolutionary chain. They start to attack the problem of data abstraction but they do not go far enough. They work, but there are problems:

1. Most importantly, there is no easy way to modify or extend the behavior of the unit after it is compiled.
2. These abstract types don't mesh with the original language very well. Syntactically they are a mess, and they don't use any of the operators like the "normal" types do. For example, if you create a new type for which an addition operation is natural, there is no way for you to use the plus sign to signify the operation--you have to create a function called **add** instead.
3. If you hide an array in a unit you can have only one array. You cannot create multiple instances of the data type unless you modify the code and break the data hiding principle in the process.

C++ classes eliminate these deficiencies.

3.3 C++ and Data Abstraction

In response to these problems, object oriented languages such as C++ offer easy, extensible ways to implement data abstraction. All that you have to do is modify your thinking patterns so that you think about problems in an "abstract" way. This mental shift is fairy easy once you have seen some examples.

First of all you want to try to think in terms of "data types". Whenever you create a data type you need to think of all of the things you will want to do with that data type and then bind the functions you create to the type. For example, say that you are creating a program which requires a rectangle data type containing two coordinate pairs. You should think, "what will I need to do with this type?" You might come up with the following actions: set it to a value, check for equality with another rectangle, check for intersection with another rectangle, and check to see if a point is inside the rectangle. If you need a terrain data type, you go through the same process and come up with functions to load the terrain data, draw the data, and so on. You then bind these functions to the data. Doing this for each data type you need in a program is the essence of object oriented program.

The other essential technique used when thinking in an object oriented way involves training your mind to think in a "generic-to-specific" hierarchy. For example, when thinking about a terrain object, you might notice some similarities between it and a list. After all, somewhere in there is a list of coordinates that is loaded from the file. A list is a generic object that can be used in many places. So you would try to create a generic list class and then build the terrain object on top of it. We will examine this process in detail as we go though more examples in the following tutorials.

Understanding C++: An Accelerated Introduction
by Marshall Brain

A Simple C++ Vocabulary

We can use a specific example to firm up some of the ideas from the last section. In this tutorial we will look at a simple address list program implemented in C, and see how it can be moved to C++ by adding a class.

4.1 An address list program

Let's say that you want to create an address list program that manages a list of names and addresses. The first thing you want to do to create this program is describe the program in English. It turns out that a good English description also helps to find objects in a program, and this is useful when designing C++ code. The description helps you to see the objects you need to create, as well as the functions that will go with each object. Here is a typical description:

I want to create an address list program. The program will hold a list of names and addresses. The user will be able to add entries to the list, print the list to the screen, and find entries in the list.

You can see that this is a very high level description. It doesn't talk about the user interface, loading and saving information on disk, error checking, the record format, or the data structure used. All of that would come later. The point of this description is to see what it *does* talk about. In particular, it talks about an object--a **list**-- and a set of actions that go with the object--adding, printing, and finding. Now let's take the description further:

The list can be loaded from disk and saved to disk. When the program begins, it will load the list and then display a menu that lets the user select from the following options: add, delete, find, and quit. When the user selects quit, the list will be saved and the program will terminate.

From this description, you can see that there are two more actions for the list object-- load and save. You can also see two new objects developing--the **menu** object and the **program** object. Two actions are listed for the menu: display and selection. The program object currently has three actions: initialization, menu display, and termination.

The point to gain from this example is that an application breaks down into objects fairly naturally. As you describe the program, you begin to see objects in the description. They are generally the nouns in the description. You also can see the functions for the object--they are the verbs. One technique for finding objects in a program is to describe it, make a list of nouns from that description, and then throw out obvious things like "the user". What's left is a set of objects that the program will have to deal with. Then make a list of verbs and use them to form functions for each object.

4.2 An old-style program

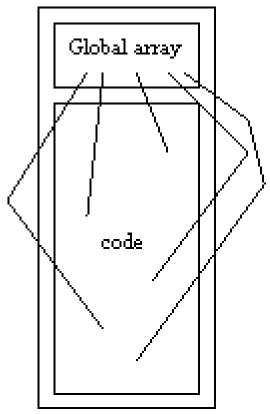
Let's start creating this address list program by implementing it in C. Then we will move it to C++ by adding a class. The following listing shows a very simple implementation of the address list using normal functions. The program can add elements to the list, print the list to the screen, or find an item in the list. The list is held in a global array.

```
#include <iostream.h>
```

```
#include <string.h>

typedef struct
{
    char name[20];
    char city [20];
    char state[20];
} addrStruct;

const int MAX=10;
addrStruct list[MAX];
int numInList;

void addName()
{
    if (numInList < MAX)
    {
        cout << "Enter Name: ";
        cin >>  list[numInList].name;
        cout << "Enter City: ";
        cin >> list[numInList].city;
        cout << "enter State: ";
        cin >> list[numInList].state;
        numInList++;
    }
    else
    {
        cout << "List full\n";
    }
}

void printOneName(int i)
{
    cout << endl;
    cout << list[i].name << endl;
    cout << list[i].city << endl;
    cout << list[i].state << endl;
}

void printNames()
{
    int i;

    for (i=0; i < numInList; i++)
        printOneName(i);
    cout << endl;
}

void findName()
{
    char s[20];
    int i;
    int found=0;

    if (numInList==0)
    {
        cout << "List empty\n";
    }
    else
    {
        cout << "Enter name to find: ";
        cin >> s;
        for (i=0; i < numInList; i++)
        {
            if (strcmp(s,list[i].name)==0)
            {
                printOneName(i);
                found=1;
            }
        }
        if (!found)
            cout << "No match\n";
    }
}

void paintMenu()
{
    cout << "Address list Main Menu\n";
    cout << "  1 - add to list\n";
    cout << "  2 - print list\n";
    cout << "  3 - find name\n";
    cout << "  4 - quit\n";
    cout << "Enter choice: ";
}

void main()
{
    char choice[10];
    int done=0;
    numInList=0;
    while (!done)
    {
        paintMenu();
        cin >> choice;
        switch(choice[0])
        {
            case '1':
                addName();
                break;
            case '2':
                printNames();
                break;
            case '3':
                findName();
                break;
            case '4':
                done=1;
                break;
            default:
                cout << "invalid choice.\n";
        }
    }
}
```

This program has a fairly typical structure and organization. Functions are used to break up the code. One function handles each of the menu options, one paints the menu, and the function **printOneName** holds a piece of redundant code used in two places in the program. This program demonstrates the two main uses for functions-- decomposition/naming and redundancy removal.

There is one fundamental problem with this program: The code is strongly bonded to the global array variable. As shown in the figure below, the array is global and it is referenced directly throughout the program:



There is no easy way to change the array to another data structure without rewriting most of the code. This code has nothing to do with the list implemented by the array--it simply is in the wrong place.

The idea behind data abstraction is to protect variables such as the global array from direct manipulation by the program. By isolating the variables implementing the list from the rest of the program with function calls, we can accomplish three things:

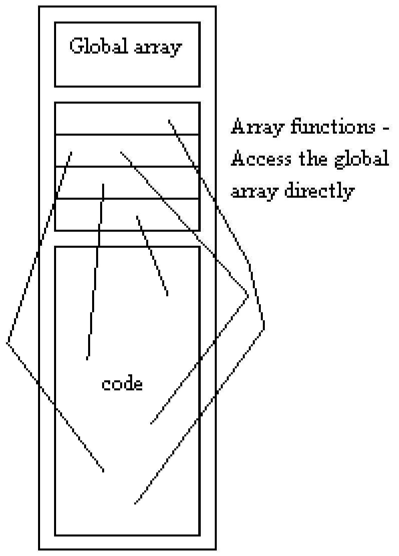
1. It is much easier to replace the list with different data structures later on, because only the list functions need changing.
2. The program is better organized--the list concept is separated out from the rest of the code as much as possible.
3. The list functionality can be used elsewhere in other programs now that it stands on its own.

In C you would make the program look like this:

```
#include <iostream.h>
#include <string.h>

typedef struct
{
    char name[20];
    char city [20];
    char state[20];
} addrStruct;

//-------- data and functions for the list -------
const int MAX=10;
addrStruct list[MAX];
int numInList;

void listInit()
{
    numInList=0;
}
```

```
void listTerminate()
{
}

int listFull()
{
    if (numInList >=MAX) return 1; else return 0;
}

int listEmpty()
{
    if (numInList==0) return 1; else return 0;
}

int listSize()
{
    return numInList;
}

int listAdd(addrStruct addr)
{
    if (!listFull())
    {
        list[numInList++]=addr;
        return 0;  // returns 0 if OK
    }
    return 1;
}

int listGet(addrStruct&  addr, int i)
{
    if (i < listSize())
    {
        addr=list[i];
        return 0;  // returns 0 if OK
    }
    return 1;
}
//-------------------------------------------------

void addName()
{
    addrStruct a;

    if (!listFull())
    {
        cout << "Enter Name: ";
        cin >> a.name;
        cout << "Enter City: ";
        cin >> a.city;
        cout << "enter State: ";
        cin >> a.state;
        listAdd(a);
    }
    else
        cout << "List full\n";
}

void printOneName(addrStruct a)
{
    cout << endl;
    cout << a.name << endl;
    cout << a.city << endl;
    cout << a.state << endl;
}

void printNames()
{
    int i;
    addrStruct a;

    for (i=0; i < listSize(); i++)
    {
        listGet(a,i);
        printOneName(a);
    }
    cout << endl;
}

void findName()
{
    char s[20];
    int i;
    int found=0;
    addrStruct a;

    if (listSize==0)
        cout << "List empty\n";
    else
    {
        cout << "Enter name to find: ";
        cin >> s;
        for (i=0; i < listSize(); i++)
        {
            listGet(a, i);
            if (strcmp(s,a.name)==0)
            {
                printOneName(a);
                found=1;
            }
        }
        if (!found)
            cout << "No match\n";
    }
}

void paintMenu()
{
    cout << "Address list Main Menu\n";
    cout << "  1 - add to list\n";
    cout << "  2 - print list\n";
    cout << "  3 - find name\n";
    cout << "  4 - quit\n";
    cout << "Enter choice: ";
}

void main()
{
    char choice[10];
    int done=0;
    listInit();
    while (!done)
    {
        paintMenu();
        cin >> choice;
        switch(choice[0])
        {
            case '1':
                addName();
                break;
            case '2':
                printNames();
                break;
            case '3':
                findName();
                break;
            case '4':
                done=1;
                break;
            default: cout << "invalid choice.\n";
        }
    }
    listTerminate();
}
```

At the top of the program are seven functions as well as the variables used to implement the list. The goal of the functions is to completely protect, or *encapsulate*, the variables. Using the **list...** functions it is possible to do anything that this program needs to do to the list without using any of the actual variables that implement the list. The functions act as a wall between the variables and the program. With this program structure, any change to the implementation of the list (for example, changing the array to a linked list) has no effect on the program itself--only the seven functions must be modified. The structure of this program is shown below:



Many of these functions may seem trivial. For example, the **listTerminate** function contains no actual code at all. But it is there because of future possibilities--if the implementation changes to a linked list, there will need to be a function that deletes all of the elements in the list to avoid memory leaks. The **listSize** function contains just one line here, but if the list were implemented using a binary tree the function would have to recursively traverse the tree to count all of the elements and it would be much larger. What we have done is think of all the functions that might actually be needed for a generic list no matter how it is implemented.

While the implementation above is successful in isolating the list from the rest of the program, it has several problems. For example, anyone could come along and modify the program, calling the variables directly and defeating the wall of functions. In other words, there is no enforcement of the wall. Also, it is not easy to use two of these lists in one program. All of the functions are tightly bound to a single array. You could get around this problem by passing the array in as a parameter, but that gets messy. C++ solves both problems with classes.

4.3 Creating a class -

The following code takes the data and the seven list functions from the previous listing and implements them as a C++ class. It then uses that class in the program:

```
#include <iostream.h>
#include <string.h>

typedef struct
{
    char name[20];
    char city [20];
    char state[20];
} addrStruct;

const int MAX = 10;

class List
{
    addrStruct list[MAX];
    int numInList;
public:
    List(): numInList(0) // constructor
    {
    }
    ~List() // destructor
    {
    }
    int Full()
    {
        if (numInList >=MAX) return 1; else return 0;
    }
    int Empty()
    {
        if (numInList==0) return 1; else return 0;
    }
    int Size()
    {
        return numInList;
    }
    int Add(addrStruct addr)
    {
        if (!Full())
        {
            list[numInList++]=addr;
            return 0;  // returns 0 if OK
        }
        return 1;
    }
    int Get(addrStruct&  addr, int i)
    {
        if (i < Size())
        {
            addr=list[i];
            return 0;  // returns 0 if OK
        }
        return 1;
    }
};
//-----------------------------------------------

List list;

void addName()
{
    addrStruct a;

    if (!list.Full())
    {
        cout << "Enter Name: ";
        cin >> a.name;
        cout << "Enter City: ";
        cin >> a.city;
        cout << "enter State: ";
        cin >> a.state;
        list.Add(a);
    }
    else
        cout << "List full\n";
}

void printOneName(addrStruct a)
{
    cout << endl;
    cout << a.name << endl;
    cout << a.city << endl;
    cout << a.state << endl;
}

void printNames()
{
    int i;
    addrStruct a;

    for (i=0; i < list.Size(); i++)
    {
        list.Get(a,i);
        printOneName(a);
    }
    cout << endl;
}

void findName()
{
    char s[20];
    int i;
    int found=0;
    addrStruct a;
    if (list.Size()==0)
        cout << "List empty\n";
    else
    {
        cout << "Enter name to find: ";
        cin >> s;
        for (i=0; i < list.Size(); i++)
        {
            list.Get(a, i);
            if (strcmp(s,a.name)==0)
            {
                printOneName(a);
                found=1;
            }
        }
        if (!found)
            cout << "No match\n";
    }
}

void paintMenu()
{
    cout << "Address list Main Menu\n";
    cout << "  1 - add to list\n";
    cout << "  2 - print list\n";
    cout << "  3 - find name\n";
    cout << "  4 - quit\n";
    cout << "Enter choice: ";
}

int main()
{
    char choice[10];
    int done=0;

    while (!done)
    {
        paintMenu();
        cin >> choice;
        switch(choice[0])
        {
            case '1':
                addName();
                break;
            case '2':
                printNames();
                break;
            case '3':
                findName();
                break;
            case '4':
                done=1;
                break;
            default:
                cout << "invalid choice.\n";
        }
    }
    return 0;
    // list destroys itself when it goes out of scope.
}
```

The list class is near the top of the program and starts with the words **class List**. This is just a type declaration--the actual *instance* of the list appears at the line:

```
List list;
```

This line declares a variable named **list** of the type **class List.**

Notice that the **List** class starts off looking very much like a structure. It declares two variables in the same way a structure would. These are called *data members*. It then contains the word "public": This word indicates that the following functions will be known to any code using this class. The opposite word is "private" and is used when functions or variables are to remain hidden from the rest of the program. The variables and functions defined in a class are by default private unless you specifically make them public as shown here (the 2 data members are private by default, and the 7 functions are public).

Following the data members come the *member functions*. These are the functions that can be applied to instances of this class. The first two functions-- **List** and **~List**--are unique, and are called the *constructor* and the *destructor* respectively. The constructor is *automatically* called when any instance of this class comes into existence. In this case, the instance comes into existence at the start of program execution because it is declared as a global variable, but constructors of local variables are called when the local variable comes into existence and constructors of pointers are activated when **new** is called on the pointer. The constructor has the same name as the class itself:

```
List(): numInList(0) // constructor
{
}
```

The initialization of the **numInList** data member is unique here. Another way to do it would be to say:

```
List() // constructor
{
    numInList = 0;
}
```

However, the first form is more efficient at run time because of the way C++ internally initializes classes. The syntax, when used as shown in this constructor, initializes the data member **numInList** to 0 and should be used whenever initializing data members in a constructor.

The destructor **~List** is called *automatically* when the instance goes out of scope or is deleted. The remaining functions look just like C functions. They are unique only in that they are tightly bound to the class variables, and can reference the class variables at any time.

The variable **list** is an instance of this class. If **list** were a plain structure it would be declared in about the same way, and it acts the same here. The variable **list** is as big as the size of its data members. The functions do not actually take up any space in each instance of the class. The syntax of the language simply allows them to be declared, and used, with instances of the class.

The instance **list** is used throughout the program. Each time something needs to be done to **list** you find the instance name **list** followed by a dot and then a function name. This again follows the syntax of a structure. The dot says, "call the member function of the class **List** on the specific instance **list**".

This may not all make immediate sense, and that's OK. The important thing to gather from this example is that all we have done is take some data--in this case an array and an integer--and the functions needed to manipulate the variables, and we have bound them together into a *class*. Now the variables cannot be directly accessed by the rest of the code. Because they are private within the class, they can be accessed only by the class's member functions and not by any other part of the program. The list object--data and functions glued together into an object--can only be accessed via the member functions.

4.4 A Simpler Example

The last example was fairly large. Let's look at a **Stack** class to review some of the concepts learned in a smaller setting.

```
#include <iostream.h>

class Stack
{
    int stk[100];
    int top;
public:
    Stack(): top(0) {}
    ~Stack() {}
    void Clear() {top=0;}
    void Push(int i) {if (top < 100) stk[top++]=i;}
    int Pop()
    {
        if (top > 0) return stk[--top];
        else return 0;
    }
    int Size() {return top;}
};

int main()
{
    Stack stack1, stack2;

    stack1.Push(10);
    stack1.Push(20);
    stack1.Push(30);
    cout << stack1.Pop() << endl;
    stack2=stack1;
    cout << stack2.Pop() << endl;
    cout << stack2.Pop() << endl;
    cout << stack1.Size() << endl;
    cout << stack2.Size() << endl;
    return 0;
}
```

This program consists of two parts: the **Stack** class and the **main** function. The class defines the **Stack** type, and two instances of this type are declared inside of **main**. Each of the instances will have its own copy of the **stk** and **top** data members, and a **sizeof** operation on each would indicate that just enough space (202 or 404 bytes, depending on the environment) is allocated for each. A class uses just as much space as a structure with the same data members would--there is no memory overhead for the member functions.

The class contains a constructor, a destructor, and four other functions, each of which is public. Because the functions are public they can be called by any instance of the class. The constructor is called when stack variables are instantiated, and the destructor is called when they go out of scope. Inside the **main** function, different calls to the other four functions are made by using the instantiation name followed by a dot followed by a function name. For example:

```
stack1.Push(10);
```

This line indicates that the value 10 should be pushed onto **stack1**. The instance **stack1** holds two pieces of data (**stk** and **top**) which contain values. This line says, "Call the function **Push** on the structure help in **stack1**--apply the statements in **Push** and the value 10 to the actual array and integer held within **stack1**. There are two completely separate stacks in this program: **stack1** and **stack2**. A statement like stack2.Push(5) means that 5 should be pushed onto the structure **stack2**.

The assignment statement midway down the **main** function is interesting. It does the same thing that an assignment between two structures would--the values of the data members of the right side are copied to the data members on the left:

```
stack2 = stack1;
```

After the assignment statement the two stacks contain the same values. This normally works fine, but if any of the data members are pointers you have to be careful. We will see a good example of this problem in Tutorial Seven.

4.5 A rectangle class

How do you decide what should be turned into an object and what shouldn't? Essentially what you do is take each little group of related data elements that you can find in a program, attach some functions to it, and make aclass. In the stack example above, the array **stk** and the integer **top** are the data elements needed by the stack. Several useful functions relate to that little data grouping (**Push**, **Pop**, **Clear**, and **Size**). Together the data and functions make a class.

Say you have to remember the coordinates for a rectangle in one of your programs. Your variables are labeled **x1**, **y1**, **x2**, and **y2**--**x1** and **y1** represent the upper left corner and **x2** and **y2** represent the lower right corner. Together they represent a rectangle. What are some useful functions that go with these values? You need to be able to initialize them (a perfect job for the constructor), and maybe it would be handy to find the area and perimeter of the rectangle. The class might look like this:

```
class Rect
{
    int x1, y1, x2, y2;
public:
    Rect(int left=0,int top=0,
        int right=0,int bottom=0):
        x1(left),  y1(top),  x2(right),  y2(bottom)
    {
    }
    ~Rect() {}
    int Height() { return (y2-y1); }
    int Width() { return (x2-x1); }
    int Area() { return Width()*Height(); }
    int Perimeter() { return 2*Width()+2*Height();}
};
```

If you simply look at a program you are building and try to find each natural grouping of data along with some functions that are useful for manipulating that data, you will go a long way toward objectifying your programs.

4.6 Class Specifics

Let's review a few of the specifics learned in this tutorial. First, each class has a constructor and a destructor. The constructor is called when an instance of the class comes into existence and the destructor is called when the instance is destroyed. The following program can help you to learn about constructors and destructors:

```
#include <iostream.h>

class Sample
{
    int num;
public:
    Sample(int i): num(i)
    {
        cout << "constructor " << num
            << " called" << endl;
    }
    ~Sample()
    {
    cout << "destructor " << num
        << " called" << endl;}
};

int main()
{
    Sample *sp;
    Sample s(1);

    cout << "line 1" << endl;
    {
        Sample temp(2);
        cout << "line 2" << endl;
    }
    cout << "line 3" << endl;
    sp = new Sample(3);
    cout << "line 4" << endl;
    delete sp;
    cout << "line 5" << endl;
    return 0;
}
```

$\boxed{\text{HW}}$

Try running this code on paper and predict what it will do. Then run the program with a single-stepping debugger and see what happens.

Data members and member functions can be public or private, depending on their role in the program. It is good to strive toward the goal of no public data members. A public member can be used anywhere in the program, while a private member can only be used by a function that is a member of the class. Let's modify the **Rect** class slightly to see what this means:

```
class Rect
{
    int x1, y1, x2, y2;
public:
    Rect(int left=0,int top=0,
        int right=0,int bottom=0):
        x1(left),  y1(top),  x2(right),  y2(bottom)
    {
    }
    ~Rect() {}
private:
    int Height() { return (y2-y1); }
    int Width() { return (x2-x1); }
public:
    int Area() { return Width()*Height(); }
    int Perimeter() { return 2*Width()+2*Height();}
};
```

Now the **Width** and **Height** functions are private. They can be called as shown here because **Area** and **Perimeter** are member functions. But if you try the following:

```
    Rect r;
    ...
    cout  << r.Height();
```

You will get a compiler error because **Height** is private.

Assignment between two instances of a class simply copies the data members from one instance to the other. For example:

```
    Rect r1,r2;
    ...
    r1=r2;
```

is the same as saying:

```
    r1.x1 = r2.x1;
    r1.y1 = r2.y1;
    r1.x2 = r2.x2;
    r1.y2 = r2.y2;
```

Finally, there are two accepted ways to specify member functions. The examples seen previously represent one method, called **inline** functions. The code below shows the second method, here applied to the **Rect** class:

```
class Rect
{
    int x1, y1, x2, y2;
public:
    // the constructor uses default param. See tutor 2
    Rect(int left=0,int top=0,int right=0,int bottom=0);
    ~Rect();
    int Height();
    int Width();
    int Area();
    int Perimeter();
};


Rect::Rect(int left, int top, int right, int bottom):
    x1(left),  y1(top),  x2(right),  y2(bottom)
// default values are understood from the prototype
{
}


Rect::~Rect()
{
}

int Rect::Height()
{
    return (x2-x1);
}

int Rect::Width()
{
    return (y2-y1);
}

int Rect::Area()
{
    return Width()*Height();
}

int Rect::Perimeter()
{
    return 2*Width()+2*Height();
}
```

This form is generally much easier to read when the functions in the class are long. The **Rect::** portion specifies the class to which the function belongs. The class definition itself contains what are essentially prototypes for the class functions.

There are many other things that you can do with a class, but to create simple data- abstracting collections of functions and data the material presented here is all that you need. Now we can start creating hierarchies from these classes.

Understanding C++: An Accelerated Introduction

by Marshall Brain

Inheritance

Let's say that you have a list class, and now you want to modify it. In the old world of programming you would take the source and start changing things. In the object oriented world of programming you do things differently. What you do instead is leave the existing class alone and then layer your changes on top of it using a process called *inheritance*. Layering through inheritance lies at the very heart of object oriented programming. It is a totally different way of doing things, but it has several important advantages:

1. Let's say that you bought the list class from someone else, so you don't have the source code. By leaving the existing class alone and layering your changes on top of it you don't *need* to have the source.

2. The existing class is completely debugged and tested. If you modify its source, it has to go through the testing process again to be re-certified. Changes you make might also have side-effects that aren't detected immediately. By layering the changes on top of the existing class, the existing class never changes and therefore remains bug-free. Only the new pieces must be tested.

3. The layering process forces you to think in a generic-to-specific way. You create a generic class like a list, and then layer specificity on top of it. A nice bonus of this way of thinking is that the generic classes are useful in many different programs. A list, for example, is useful in a lot of places. Each new program layers its own specifics onto the generic list, but the generic list stays the same everywhere.

4. If the "base class" is improved, all classes built on top of it take advantage of those improvements without modification. For example, say that the list class is changed so that it sorts 10 times faster than it used to. Now every class built on top of the list class sorts 10 times faster as well, without modifying anything.

It is these benefits that get people excited about object oriented programming.

5.1 Inheritance example

Let's look at a specific example to get a feel for how inheritance works. Say you have purchased a simple list manager. It has the ability to insert at a specified location, to get items from the list, and to return the size of the list. The code for this list class is shown below, along with a small piece of test code:

```
#include <iostream.h>

class List
{
    int array[100];
    int count;
public:
    List(): count(0) {}
    ~List() {}
    void Insert( int n, int location )
    {
        int i;
        for (i=count; i >= location; i--)
            array[i+1] = array[i];
        array[location]=n;
        count++;
    }
    int  Get( int location ) {return array[location];}
    int Size() { return count; }
};

void main()
{
    List list;
    int i, value;

    for (i=0; i < 10; i++)
        list.Insert(i,i);
    list.Insert(100,5);
    list.Insert(200,7);
    list.Insert(300,0);
    for (i=0; i < list.Size(); i++)
        cout << list.Get(i) << endl;
}
```

The class contains no error checking to keep it small--obviously you would want to add some if this were a commercial product.

Now let's say that you want to modify this class to add two features. First, you want to have a sorted insertion function so that the class maintains a sorted list. Second, you want to keep track of the total sum of all items in the list. Rather than cycling through all elements in the list each time the **sum** function is called, you want to keep a running total as each item is inserted.

Obviously one way to do this is to simply modify the **List** class shown above. In C++ you use inheritance to make the changes instead. We will create a **SortedList** class by inheriting the **List** class and modifying it. Let's start by adding the sorted insertion feature:

```
class SortedList: public List
{
public:
    SortedList():List() {}

    void SortedInsert(int n)
    {
        int i,j;

        i=0;
        do
        {
            j = Get(i);
            if (j < n ) i++;
        } while (j < n && i < Size());
        Insert(n, i);
    }
};
```

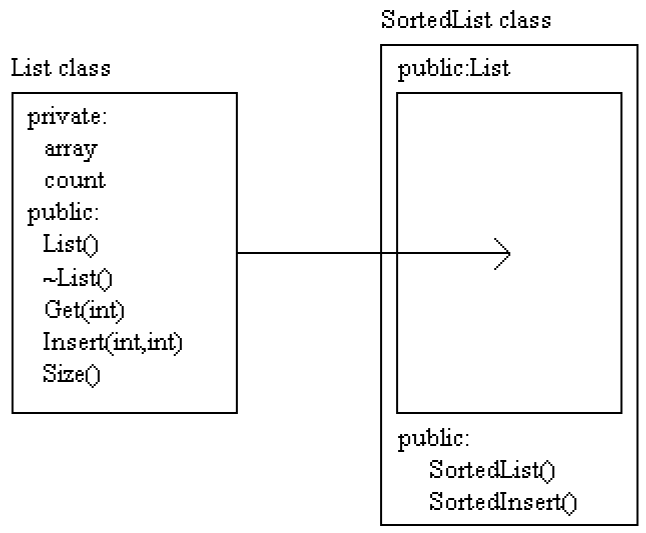
The **List** class is totally unchanged--we have simply created the **SortedList** class on top of it. The **SortedList** class *inherits* its behavior from the **List** class--it is *derived* from the **List** class. The **List** class is the *base class* for **SortedList**.

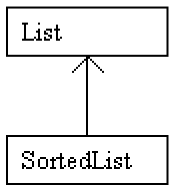
The **List** class is inherited on the fist line:

```
class SortedList: public List
```

The colon indicates that we are inheriting something. The word **public** indicates that we want the public functions and variables in **List** to remain public in the **SortedList** class. We could have also used **private** or **protected**. In either of these cases any **public** variables and functions in the inherited class would be converted in the derived class. The use of **public** here is standard.

The diagram below shows what is happening:





The **SortedList** class simply extends the **List** class. Anyone using the **SortedList** class has access to the functions available in **List** as well as the new functions available in **SortedList**.

The constructor for **SortedList** is also new--we have used a colon here to call the constructor for the inherited class:

```
SortedList():List() {}
```

This line says that the constructor named **List** from the base class should be called, and that the **SortedList** constructor needs to do nothing of its own.

In the remainder of the **SortedList** class we simply add the new **SortedInsert** function into the class. This new function makes use of the old **Insert**, **Get**, and **Size** functions from the **List** class as needed, but it does not access any of the **List** class data members directly because it can't--they are private to the **List** class, so they cannot be seen in the inheriting class.

Say that you wanted to have a variable or a function that seems private to outside users of a class but seems public to classes that inherit the class. For example, say that the **SortedList** class needed direct access to the **array** variable in **List** in order to improve its performance, but we still want to keep normal instances of **List** and **SortedList** from accessing the array directly.

The word **protected:** can be used in the same manner as **public:** or **private:** to indicate this behavior. By declaring **array** as a protected member in **List**, it would be accessible by the derived class **SortedList** but not by normal instances of **List** or **SortedList**.

Now let's add the totaling capability to the **SortedList** class. To do this we will need to add a new variable, and we will also need to modify the **Insert** function so that each insertion adds to the total. The code is shown below:

```
class SortedList: public List
{
private:
    int total;
public:
    SortedList():List(), total(0) {}
    void Insert( int n, int location )
    {
        total = total + n;
        List::Insert(n, location);
    }
    int GetTotal() { return total; }
    void SortedInsert(int n)
    {
        int i,j;
        i=0;
        do
        {
            j = Get(i);
            if (j < n ) i++;
        } while (j < n && i < Size());
        Insert(n, i);
    }
};
```

In this version of the **SortedList** class we have added a new data member named **total**, a new member function **GetTotal** to retrieve the current total, and a new function **Insert** which *overrides* the existing **Insert** function. We have also modified the **SortedList** constructor so that it initializes **total**. Now whenever the **SortedList** class is used and the **Insert** function is called, the *new* version of the **Insert** function will be accessed instead of the old version in **List**. The same goes for the **SortedInsert** function as well--when it calls **Insert** it is calling the *new* version.

The code for the new **Insert** function is straightforward:

```
    void Insert( int n, int location )
    {
        total = total + n;
        List::Insert(n, location);
    }
```

This function first adds the new value to the total. It then calls the *old* **Insert** function inherited from the base class so that the value is inserted in the list properly. The **List::** specifies from which class in the hierarchy the **Insert** function should be chosen. This is only a two-level hierarchy so it is a simple decision here, but in a hierarchy that has several layers of inheritance you can use this technique to choose a specific function from many. It is this layering, and the ability to work and think in a multi-level inheritance hierarchy as shown here, that gives C++ its 3- dimensional feel.

5.2 A More Advanced Example

Let's take what we have learned about inheritance and use it to create a realistic example class. What we would like to do is create a new number class called a "multi- precision integer", or "mint". This integer type will work like a normal integer, but it will have up to 100 digits (for now--later we will see how to extend it to have as many digits as memory will hold using linked lists). A mint allows you to do things like find the actual value for 60!, or find the 300th value in a Fibonacci sequence.

Visualizing Class Hierarchies

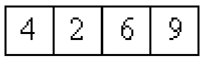
Whenever you begin using inheritance you have to start creating **Class hierarchies**. One of the most frusterating things when you are first using class hierarchies is the "Where am I?" feeling you get. Many larger hierarchies have *hundreds* of classes. A good way to get around this feeling is to use a class hierarchy visualization tool like CodeVizor. With CodeVizor you can drag the source code into the CodeVizor tool and in about 30 seconds have a beautiful, clickable (and printable!) class hierarchy chart. You can even color classes individually or in groups so that they stand out! Get CodeVizor and see how much easier undestanding class hierarchies becomes!

CodeVizor Web Site

What is a good way to create the new class in a object-oriented programming environment? One way to think about it is to think in a generic-to-specific way. For example, what is a multi-precision integer? It is simply a list of digits. Therefore, you can start by creating a generic list class that has all of the insertion features needed to implement a mint, and then layer the mint functionality on top of it.

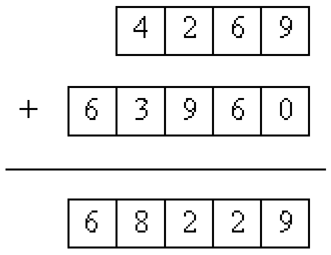
How do we decide which features are needed in the list? A good way to do this is to think about what you will have to do with the digits in typical mint operations, and then use those thoughts to create the list class. Alternatively, you would have a list class laying around and you would simply build on top of it. Let's take the first approach since we don't have a good list class laying around.

How do you initialize a mint? The mint will start off containing no digits. We will then add one digit at a time to create the new mint. For the value 4,269 the mint would look like this:



Each square in this diagram represents one element in the list, and each element in the list contains an integer value between 0 and 9. At the list level we need to be able to add digits to the beginning or the end of the list, depending on where the initial value came from.

Now let's look at a simple addition, as shown in the figure below:



In order to implement addition we will want to start with the last digits of the two mints being summed, add them together, and insert the resulting digit in the new mint being formed as the sum. Then we will go to the previous two digits and do the same thing, and so on. We will therefore need an efficient way to move through the lists from end to beginning (for example, **GetLast** and **GetPrevious** functions), and we will also need a way to be able to tell when we have hit the beginning of the list (perhaps a return value from **GetPrevious** can indicate that the action is not possible, or a **Size** function can indicate how far to go).

From this discussion and our previous work with lists we can surmise that the list will probably need to have the following capabilities:

- constructor and destructor
- AddToFront
- AddToEnd
- GetFirst
- GetLast
- GetPrevious
- GetNext
- Size
- Clear

The code below implements the list:

```
class List
{
    int array[100];
    int count;
    int pointer;
public:
    List(): count(0), pointer(0) {}
    ~List() {}
    void AddToFront(int n)
    {
        int i;
        for(i=count; i >= 1; i--)
            array[i]=array[i-1];
        array[0]=n;
        count++;
    }
    void AddToEnd(int n)
    {
        array[count++]=n;
    }
    // &n is a reference - see tutor 2
    int GetFirst(int & n)
    {
        if (count==0)
            return 1;
        else
        {
            n=array[0];
            pointer=0;
            return 0;
        }
    }
    int GetLast(int & n)
    {
        if (count==0)
            return 1;
        else
        {
            n=array[count-1];
            pointer=count-1;
            return 0;
        }
    }
    int GetPrevious(int & n)
    {
        if (pointer-1 < 0)
            return 1;
        else
        {
            pointer--;
            n=array[pointer];
            return 0;
        }
    }
    int GetNext(int & n)
    {
        if (pointer+1 > count-1)
            return 1;
        else
        {
            pointer++;
            n=array[pointer];
            return 0;
        }
    }
    int Size() { return count; }
    void Clear() { count = 0; }
};
```

This code should all be fairly straightforward to you at this point. **List** is simply a generic list of integers. A data member named **pointer** points to one of the elements in the list and is moved by the four **Get...** functions. Each of these functions returns 0 on success and 1 on failure (for example, if **pointer** is not on element 0 of the list then there is a previous element to get and **GetPrevious** function will return a 0). The two **Add...** functions add at the beginning and end of the list respectively--they currently contain no error checking. The **AddToFront** function contains an inherent inefficiency because it must move the entire contents of the array down one element for each insertion.

The **Mint** class inherits **List** and uses it to build the actual mint type. It implements two constructors (a *default constructor* that accepts no parameters and a second constructor that accepts a string and uses it to fill the list), as well as functions that add two mints and print a mint. The code is shown below:

```
class Mint: public List
{
public:
    Mint():List() {}
    Mint(char *s):List()
    {
        char *p;
        for (p=s; *p; p++)
```

```
            AddToEnd(*p-'0');
    }
    void Add(Mint & a, Mint & b)
    {
        int carry, temp;
        int erra, errb, na, nb;

        carry=0;
        Clear();
        erra=a.GetLast(na);
        errb=b.GetLast(nb);
        while (!erra || !errb)
        {
            if (erra)
                temp=nb+carry;
            else if (errb)
                temp=na+carry;
            else
                temp=na+nb+carry;
            AddToFront(temp%10);
            carry=temp/10;
            erra=a.GetPrevious(na);
            errb=b.GetPrevious(nb);
        }
        if (carry > 0)
            AddToFront(carry);
    }
    void Print()
    {
        int n, err;

        err=GetFirst(n);
        while( !err )
        {
            cout << n;
            err=GetNext(n);
        }
        cout << endl;
    }
};
```

The following **main** function tests the mint class by adding two numbers and printing the sum:

```
void main()
{
    Mint a("1234567");
    Mint b("1234");
    Mint c;

    c.Add(a,b);
    c.Print();
}
```

The constructors and the **Print** function are simple and straightforward. The **Add** function may remind you of your grade school days, because it is doing addition the old fashioned way. It starts with the last digits of the two numbers being summed, adds those digits, saves the result in the current mint, and remembers the carry value. It then moves forward through the list. Since it is likely that the two mints will not have an equal number of digits, the code must continually check to make sure that it has not run out of digits in one or the other mint. It does this using **erra** and **errb**. As soon as both mints have run out it checks **carry** and saves one last digit if necessary.

Running the test code you will see that the **Mint** class works as advertised and can add two numbers of up to 100 digits each. After you use the **Mint** class for awhile though you begin to see a problem with the **Add** function--there is no way to say something like "m = m + 1", or in the format necessary here "m.Add(m, one);" where **one** has been initialized to "1". The reason for this lies in the fact that **Add** must clear out the destination of the result before it can place a value into it, and this forces the loss of needed data in the case shown here.

The solution to this problem lies in the creation of a temporary holding value for the result during the actual addition. Then at the end of the function, the final result is copied into the current instance. The **this** pointer is used to solve the problem, as shown below:

```
void Add(Mint & a, Mint & b)
{
    int carry, temp;
    int erra, errb, na, nb;
    Mint x;

    carry=0;
    erra=a.GetLast(na);
    errb=b.GetLast(nb);
    while (!erra || !errb)
    {
        if (erra)
            temp=nb+carry;
        else if (errb)
            temp=na+carry;
        else
            temp=na+nb+carry;
        x.AddToFront(temp%10);
        carry=temp/10;
        erra=a.GetPrevious(na);
        errb=b.GetPrevious(nb);
    }
    if (carry > 0)
        x.AddToFront(carry);
*this = x;
}
```

In this version of **Add** a temporary value named **x** has been created. The results of the addition are placed into **x** digit by digit. The last line of the function copies **x** into the current instance. The **this** pointer is available in every instance of a class in C++--it points to the current instance. That is, **this** is a pointer that points to the data members (the structure) that make up the current instance. In this case we use **this** because it saves code. The alternative would be to replace the last line with:

```
array = x.array;
count = x.count;
pointer = x.pointer;
```

The value ***this** is the structure pointed to by **this**, and it is more expedient to copy the whole structure at once.

As a final example of the **Mint** class, let's use it to implement a Fibonacci number finder. The Fibonacci sequence is as follows:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, etc.
```

Each number in the sequence is the sum of the prior two numbers. In order to implement this feature we will need a way to check for equality in mints so that we can make a loop. The following member function can be added to the **Mint** class to check for equality between two mints:

```
    int Equal(Mint & a)
    {
        if (a.Size()!=Size())
            return 0;
        else
        {
            int i, na, nb;
            a.GetFirst(na);
            GetFirst(nb);
            for (i=0; i < a.Size(); i++)
                if (na!=nb)
                    return 0;
                else
                {
                    a.GetNext(na);
                    GetNext(nb);
                }
            return 1;
        }
    }
```

Given the existence of this function, then the following code will find the 100th number in the Fibonacci sequence:

```
void main()
{
    Mint max("100");
    Mint counter("1"), one("1");
    Mint t1("0"), t2("1");
    Mint d;

    do
    {
        d.Add(t1,t2);
        t1=t2;
        t2=d;
        counter.Add(counter,one);
    } while (!counter.Equal(max));
    d.Print();
}
```

The code uses two values **t1** and **t2** to remember the previous two values. They are added together and then shifted down by one. The counter is then incremented and the loop continues until the counter has reached the desired value. Using this code, the 100th number was found to be 354,224,848,179,261,915,075.

5.3 Conclusion

In this tutorial you have seen how inheritance is used to create class hierarchies, and at how the existence of inheritance tends to favor the development of code using a generic-to- specific style. The **Mint** class is a perfect example of this phenomena--a generic list was used to build the **Mint** class because a mint is nothing more than a list of digits.

Although we have accomplished our goal, the **Mint** class is not very well integrated into the language. We would like to use the "+" operator for addition and the "==" operator to check for equality. We will see how to do this in the next section.

Understanding C++: An Accelerated Introduction

by Marshall Brain

Operator Overloading

In the last tutorial we implemented a version of the **Mint** class, ending up with code that calculates members of the Fibonacci sequence. The code used to perform the calculation looked like this:

```
void main()
{
    Mint max("100");
    Mint counter("1"), one("1");
    Mint t1("0"), t2("1");
    Mint d;

    do
    {
        d.Add(t1,t2);
        t1=t2;
        t2=d;

        counter.Add(counter,one);
    } while (!counter.Equal(max));
    d.Print();
}
```

What we would like instead is to be able to write code that looks "normal", like this:

```
void main()
{
    Mint max("100");
    Mint counter("1");
    Mint t1("0"), t2("1");
    Mint d;

    do
    {
        d = t1 + t2;
        t1=t2;
        t2=d;
        counter = counter + "1";
    } while (! (counter==max));
    cout << d << endl;
}
```

C++ allows this sort of seamless melding of new types using a process called *operator overloading*. The normal operators like "+", "==", and "<<" are overloaded so that they can handle the new types.

Some operator overloading involves the use of *friend* functions. A friend function is just like a normal C function, but it is permitted to access private members of the class within which it is declared. The fact that it is a normal C function means that it does not have access to a **this** pointer, and also that it can be called without having to name a class that it operates on. For example, a normal member function such as **Insert** in the **List** class requires an instantiation of the list to be called:

```
List lst;
...
lst.Insert(5);
```

A friend function does not necessarily require a class instantiation because it does not have a **this** pointer.

Almost every operator in C++ can be overloaded:

```
    +     -     *     /     %     ^     &     |
    ~     !     ,     =     <     >     <=    >=
    ++    --    <<    >>    ==    !=    &&    ||
    +=    -=    /=    %=    ^=    & =   |=    *=
    <<=   >>=   [ ]   ( )   ->    ->*   new   delete
```

Many of these are never seen, much less overloaded, but by overloading all of the common operators like "+" and "==" you can make a class much easier to use.

The code below shows the **Mint** class redone so that the "+", "==", and "<< " operators are overloaded, along with a piece of test code that uses all three:

```
class Mint: public List
{
public:
    Mint():List() {}
    Mint(char *s):List()
    {
        char *p;
        for (p=s; *p; p++)
            AddToEnd(*p-'0');
    }

    friend Mint operator+ (Mint & a, Mint & b)
    {
        int carry, temp;
        int erra, errb, na, nb;
        Mint x;

        carry=0;
        erra=a.GetLast(na);
        errb=b.GetLast(nb);
        while (!erra || !errb)
        {
            if (erra)
                temp=nb+carry;
            else if (errb)
                temp=na+carry;
            else
                temp=na+nb+carry;
            x.AddToFront(temp%10);
            carry=temp/10;
            erra=a.GetPrevious(na);
            errb=b.GetPrevious(nb);
        }
        if (carry> 0)
            x.AddToFront(carry);
        return x;
    }

    int operator==(Mint & a)
    {
        if (a.Size()!=Size())
            return 0;
        else
        {
            int i, na, nb;
            a.GetFirst(na);
            GetFirst(nb);
            for (i=0; i < a.Size(); i++)
                if (na!=nb)
                    return 0;
                else
                {
                    a.GetNext(na);
                    GetNext(nb);
                }

            return 1;
        }
    }

    friend ostream&  operator << (ostream&  s, Mint & m)
    {
        int n, err;

        err=m.GetFirst(n);
        while( !err )
        {
            s << n;
            err=m.GetNext(n);
        }
        return s;
    }
};

void main()
{
    // add two numbers
    Mint a("1234567");
    Mint b("1234");
    Mint c;

    c = a + b;
    cout << "it's fine " << c << "...really" << endl;
    cout << a + "3333" << endl;

    // find the 100th Fibonacci number
    Mint counter;
    Mint t1, t2;
    Mint d;

    t1 = "0";
    t2 = "1";
    counter = "1";
    do
    {
        d = t1 + t2;
        t1 = t2;
        t2 = d;
        counter = counter + "1";
    } while (! (counter == "100") );
    cout << d << endl;
}
```

Let's start by looking at the "==" function:

int operator== (Mint & a)

Because this function is a member of the **Mint** class, this header says that the operator should return an integer, use what's on the left side of the == as **this**, and use what is on the right hand side of the == as **a**. In the code for the == operator function, when we use a function like **GetFirst** directly we are referring to the value on the left side of the ==. A function call of the form **a.GetFirst** refers to the right side of the ==:

```
Mint b, m;
...
if (b == m)
```

The rest of the code is identical to the **Equal** function that we saw in Tutorial 5. The returned integer value is used as the result of the comparison. With this function in place, our "==" operator is called whenever the compiler finds and "==" operator between two values of type **Mint**.

The oveloaded "+" operator is a friend function:

```
friend Mint operator+ (Mint & a, Mint & b)
```

It is declared as a friend because we do not want it to automatically use the left side of a plus statement as **this** because that would clear it (as discussed in Tutorial 5). Since it is a friend it acts as a normal C function without a **this** pointer. It adds the two mints passed and returns the resulting mint.

In the **main** function there are several statements of the following form:

```
c = "3333"
```

and

```
c = c + "1";
```

How does the compiler know what to do? How does it know to convert "1" to a mint? Since we have a mint constructor that accepts a **char*** type, the constructor is automatically invoked in an attempt to make the + operator's types match up. If we created another constructor that accepted a **long** parameter, then we would also be able to write code like this:

```
c = c + 1;
```

The conversion of the integer value would be automatic as well. The following statement will *not* work:

```
c = "2222" + "3333";
```

The compiler does not have anything to tell it that the "+" should be adding mints, so it cannot make the conversion--one side of the "+" must be a mint to cue the compiler.

The << operator is also overloaded. The function must be a friend because the left parameter is not of the class type. It must accept a reference to a **ostream** parameter and then to a parameter of the class type. It must also return a reference to **ostream**. Having done this however the code is simple. With this function in place any C++ output operation using a mint will work.

The >> operator is overloaded in a similar way:

```
friend istream& operator >> (istream& s, Mint&  m)
{
    buf[100];

    s >> buf;
    m = buf; // calls the constructor
    return s;
}
```

Other operators such as ++, +=, !=, etc. are easily overloaded using the examples above. For some of the more esoteric operators, see a book such as Lippman's.

Understanding C++: An Accelerated Introduction

by Marshall Brain

Working with Pointers

When a class contains data members that are pointers there are several concerns that must be addressed in order to make the class "work". For example, when an instantiation of the class is destroyed, the destructor should make sure that all allocated blocks of memory within the class are deleted. Another example involves the assignment operator: the standard "copy all data members" behavior for the "=" operator that we have seen until now has worked fine, but it does not work with pointers.

To get a feel for the differences let's implement a stack class both with an array and with pointers. Here is the array version, along with a main function containing test code (this is identical to the code we saw in Tutorial 4):

ON YOUR OWN

```
#include <iostream.h>

class Stack
{
    int stk[100];
    int top;
public:
    Stack(): top(0) {}
    ~Stack() {}
    void Clear() {top=0;}


    void Push(int i) {if (top < 100) stk[top++]=i;}
    int Pop()
    {
        if (top > 0) return stk[--top];
        else return 0;
    }
    int Size() {return top;}
};

void main()
{
    Stack stack1, stack2;

    stack1.Push(10);
    stack1.Push(20);
    stack1.Push(30);
    cout << stack1.Pop() << endl;
    stack2=stack1;
    cout << stack1.Size() << endl;
    cout << stack2.Size() << endl;
    cout << stack2.Pop() << endl;
    cout << stack2.Pop() << endl;
}
```

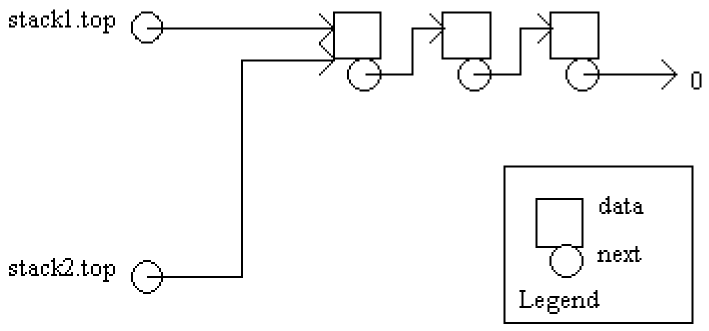
The code below implements the same stack using pointers, but it has several problems that will be discussed in a moment:

```
typedef struct node
{
    int data;
    node *next;
} node;

class Stack
{
    node *top;
public:
    Stack(): top(0) {}
    ~Stack() { Clear(); }
    void Clear()
    {
        node *p=top;
        while (p)
        {
            top = top->next;
            delete p;
            p = top;
        }
    }
    void Push(int i)
    {
        node *p = new node;
        p->data = i;
        p->next = top;
        top = p;
    }
    int Pop()
    {
        if (top != 0)
        {
            int d = top->data;
            node *p=top;
            top = top->next;
            delete p;
            return d;
        }
        else return 0;
    }
    int Size()
    {
        int c=0;
        node *p=top;
        while (p)
        {
            c++;
            p = p->next;
        }
        return c;
    }
};
```

This is a fairly complete class. It properly cleans up after itself in its destructor and works the same way as the previous stack class. However, this class does not work as expected after an assignment statement such as:

```
stack1 = stack2;
```

The following diagram demonstrates what is happening. When the assignment operation executes, it simply copies the data members from **stack2** to **stack1** leaving one copy of the data on the heap with two pointers accessing it:



After the assignment, the pointers **stack1.top** and **stack2.top** both point to the same chain of memory blocks. If one of the stacks is then cleared, or if one executes a **Pop**, the other pointer will be pointing to memory that is no longer valid. On many machines the code will compile fine and everything will look OK for awhile during execution. But as the system runs the rot sets in and things gets flakier and flakier for no apparent reason until the program finally crashes.

What is needed is a way to redo the assignment operation to create a copy of the memory blocks. But where is the assignment operator coming from, and how can it be modified?

7.1 Default Functions

Whenever you create any class, four *default* functions are created automatically unless you override them by creating your own. They are:

- The default constructor
- The default copy constructor
- The default assignment operator
- The default destructor

The default constructor is invoked whenever you declare a instance of a class and pass it no parameters. For example, if you create a class **Sample** and you create no constructors for it, then the following statement invokes the default constructor on **s**:

```
Sample s;
```

The following initialized declaration of **s2** invokes the copy constructor:

```
Sample s1;

Sample s2 = s1;
```

The default destructor is called whenever a variable goes out of scope, and the default assignment operator is called whenever a normal assignment occurs. You can override any of the defaults by creating functions of your own. For example, if you create *any* constructor then the default constructor is not created.

The code below can be used to gain an understanding of what the default constructor and destructor do:

```
#include <iostream.h>

class Class0
{
    int data0;
public:
    Class0 () { cout << "class0 constructor" << endl; }
    ~Class0 () { cout << "class0 destructor" << endl; }
};

class Class1
{
    int data1;
public:
    Class1 () { cout << "class1 constructor" << endl; }
    ~Class1 () { cout << "class1 destructor" << endl; }
};

class Class2: public Class1
{
    int data2;
  Class0 c0;
};

void main()
{
    Class2 c;
}
```

The class **Class2** has neither constructor nor destructor, but when you run this code the following output is produced:

```
class1 constructor
class0 constructor
class0 destructor
class1 destructor
```

What has happened is that the compiler created a default constructor and destructor for **Class2**. The behavior of the default constructor is to call the base class default constructor as well as the default constructor for all data members that are classes. The default constructor calls the destructors for the base class and class data members.

Let's say that you create a new constructor for **Class2** that accepts an integer. The compiler will still call the necessary default constructors for the base class and class data members. The following code demonstrates the process:

```
class Class2: public Class1
{
    int data2;
    Class0 c0;
public:
    Class2(int i)
    {
        cout << "class2 constructor" << endl;
    }
};

void main()
{
    Class2 c(1);
}
```

```
This also works, producing the following output:

class1 constructor
class0 constructor
class2 constructor
class0 destructor
class1 destructor
```

But now you cannot declare an uninitialized variable of type **Class2** because there is no default constructor. The following code demonstrates:

```
Class2 c(1);    // OK
Class2 e;    // not OK--no default constructor
```

It is also impossible to declare arrays of a class unless there is no default constructor defined. Therefore, you should recreate the default constructor yourself by creating a constructor with an empty parameter list whenever you create other constructors.

The assignment operator and copy constructor are created automatically as well. Both simply copy the data members from the right side of the equal sign to the left. In the case of our stack class we want to eliminate these default functions and use our own so that assignment works correctly. Below are the two new functions for the stack class, along with a function **Copy** that is shared by both:

```
    void Copy(const Stack&  s)
    {
        node *q=0;
        node *p=s.top;

        while (p)
        {
            if (top==0)
            {
                top = new node;
                q=top;
            }
            else
            {
                q->next = new node;
                q = q->next;
            }

            q->data = p->data;
            p = p->next;
            q->next=0;
        }
    }
    Stack&  operator= (const Stack&  s) //assignment
    {
        if (this == & s)
            return *this;
        Clear();
        Copy(s);
        return *this;
    }
    Stack(const Stack&  s): top(0) // copy constructor
    {
        Copy(s);
    }
```

The function for the assignment operator starts by checking for the case of equivalent assignment, as in:

```
s = s;
```

If it finds this situation it does nothing. It then clears the recipient and copies the linked list on the heap so that the left side of the assignment has its own copy of the stack. The copy constructor is just like any other constructor, and it is used to handle the following cases:

```
Stack s1;
s1.Push(10);
s1.Push(20);
Stack s2(s1);        // copy constructor invoked
Stack s3 = s1;    // copy constructor invoked
```

With the assignment operator and copy constructor in place, the **Stack** class is complete--it can handle any condition that may arise.

7.2 Conclusion

This may all seem like a lot of work to go through, but generally it is only necessary when working with pointers. What is happening is that you are having to actually secure your pointer-based structures against any contingency so that the data is *always* valid. In many C programs the programmer will make an assumption such as, "I can point several pointers at the same blocks on the heap, and it will be OK because in this part of the code nothing modifies the blocks." However, if another programmer comes along and violates that assumption accidentally, the program can break in mysterious and hard-to-track ways. That can never happen with a secure C++ class, because all of the contingencies are covered.

You can see that the implementation shown above is inefficient however. What if, in certain places, you *want* to have only one copy of the blocks on the heap. For example, what if the data on the heap occupies many megabytes, and you can't afford to make a copy? What you can do in that case is use a technique such as a reference count--each instance increments a static global variable that keeps count of the number of instances using the single copy of the data on the heap. Then in each destructor you can decrement the counter. Only when a destructor, after decrementing the counter, detects that no other instance is using the data in the heap does it actually delete all of the heap blocks containing the data.

Understanding C++: An Accelerated Introduction
by Marshall Brain

Virtual Functions

In these tutorials we have seen many examples of inheritance, because inheritance is very important to object oriented programming. We have seen that inheritance allows data members and member functions to be added in the derived class. We have also seen several examples where we used inheritance to *change* the behavior of a function. For example, in Tutorial 3 we saw an example where the **Insert** function of a base **List** class was overridden to implement a totaling feature. A similar hierarchy is shown below, using a base class called **List** and a derived class called **TotalingList**:

```
#include <iostream.h>

class List
{
    int array[100];
    int count;
public:
    List(): count(0) {}
    void Insert(int n) { array[count++]=n; }
    int Get(int i) { return array[i]; }
    int Size() { return count; }
};

void ManipList(List list)
{
    // do things to the list
    list.Insert(100);
    list.Insert(200);
    // do things to the list
}

class TotalingList: public List
{
    int total;
public:
    TotalingList(): List(), total(0) {}
    void Insert(int n)
    {
        total += n;
        List::Insert(n);
    }
    int GetTotal() { return total; }
};

void main()
{
    TotalingList list;
    int x;

    list.Insert(10);
    list.Insert(5);
    cout << list.GetTotal() << endl;
    ManipList(list);
    cout << list.GetTotal() << endl;
    for (x=0; x < list.Size(); x++)
        cout << list.Get(x) << ' ';
    cout << endl;
}
```

In this code, the class **List** implements the simplest possible list with the three member functions **Insert**, **Get**, and **Size** as well as the constructor. The function **ManipList** is an example of some arbitrary function that uses of the **List** class, and it calls the insert function twice simpy as an example.

The **TotalingList** class inherits the **List** class and adds in a data member named **total**. This member holds the current total of all the numbers held in the list. The **Insert** function is overridden so that **total** is updated at each insertion.

The **main** function declares an instance of the **TotalingList** class. It inserts 10 and 5, and prints out the total. It then calls **ManipList**. It might surprise you that this actually compiles--if you look at the prototype for **ManipList** you can see that it expects a parameter of type **List**, not **TotalingList**. But C++ understands certain things about inherited classes, one of them being that a parameter of a base class type should accept any class derived from that base class as well. Therefore, since **TotalingList** is derived from the **List** class, **ManipList** will accept it. This is one of the features of C++ that makes inheritance so powerful--you can create derived classes and pass them to existing functions that know only about the base class.

When the code shown above runs however, it does not produce the correct result. It produces the output:

```
15
15
10 5
```

This output indicates that not only did the totaling not work, but the 100 and 200 were never inserted in the list during the call to **ManipList**. Part of this problem is occurring because of an outright error in the code--the parameter accepted by **ManipList** must be a pointer or a reference or no values are returned. Modifying the prototype for **ManipList** to the following partially fixes the problem:

```
void ManipList(List&  list)
```

Now the output looks like this:

```
15
15
10 5 100 200
```

It is educational to single-step through the **ManipList** and watch what happens. When the calls to the Insert functions occur, they route themselves to **List::Insert** rather than **TotalingList::Insert**.

This problem can also be solved however. It is possible in C++ to create a function with the prefix **virtual**, and this causes C++ to call the version of the function *in the derived class.* That is, when a function is declared as virtual, the compiler can call versions of the function that did not even exist when the code calling the function was written. To see this, add the word **virtual** in front of the **Insert** functions in both the **List** and **TotalingList** classes, as shown below:

```
class List
{
    int array[100];
    int count;
public:
    List(): count(0) {}
    virtual void Insert(int n) { array[count++]=n; }
    int Get(int i) { return array[i]; }
    int Size() { return count; }
};

void ManipList(List&  list)


{
    // do things to the list
    list.Insert(100);
    list.Insert(200);
    // do things to the list
}

class TotalingList: public List
{
    int total;
public:
    TotalingList(): List(), total(0) {}
    virtual void Insert(int n)
    {
        total += n;
        List::Insert(n);
    }
    int GetTotal() { return total; }
};
```

Actually it is only necessary to place it in front of the function name in the base class, but its a good habit to perpetuate it in all derived classes as well to give some indication of what is happening.

Now when you execute the program, you will get the correct output:

```
15
315
10 5 100 200
```

What is happening? The word **virtual** in front of a function tells C++ that *you plan to create new versions of this function in derived classes*. That is, it lets you state future intentions for a class. When the virtual function is called, C++ looks at the class that called the function and picks the version of the function *for that class*, even if the derived class did not exist at the time that the function call was written.

What all of this means is that in many cases you have to think into the future when you are writing code. You have to think, "will I or anyone else ever need or want to change the behavior of this function?" If the answer is yes then the function should be declared as a virtual function.

You have to pay attention to several things in order for virtual functions to work correctly. For example, you have to actually predict the need for the function and remember to make it virtual in the base class. Another point can be seen in the program above--try removing the **&** from the parameter in the **ManipList** function and then single-step through the code. Even though the

Insert function is tagged as virtual, the **List::Insert** function is called instead of the **TotalingList::Insert** function. The behavior changes because the parameter type **List** is acting like a type cast when the **&** is not there. Any class passed in is cast back to the base **List** class. With the **&** in place, this casting does not happen.

You see virtual functions everywhere in C++ class hierarchies. A typical hierarchy *expects* you to be changing behavior in the future to customize the library to your application. Virtual functions are also frequently used when the creator of the class *cannot* know what you will do with the class. For example, say that you are using a user interface class that implements buttons on the screen. When you create an instance of the button it paints itself onto the screen and behaves as a button should by highlighting itself when the button is clicked by the user. However, the person who wrote the class has no idea what people using the class plan to have the button do when it is clicked. In such cases, the author will create a virtual function named something like **handleEvent** that is called whenever the button is clicked. Then you override that virtual function with a function of your own that handles the button event properly.

Conclusion

We have covered quite a bit of ground in these tutorials, but you are probably left with the impression that we have only scratched the surface. And that is true to a certain degree- -C++ is a very deep language, with many subtleties and quirks that are only mastered with experience. C is like that, only on a much smaller scale.

The only way to fully understand this language is to write, and read, a lot of C++ code. You can learn a great deal by using and studying class libraries that other people have developed.

The many advantages of this language become apparent once it is fully understood. So start coding....