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.