Appendix A

Why C++ succeeds

Part of the reason C++ has been so successful is that the goal was not just to turn C into an OOP language (although it started that way), but also to solve many other problems facing developers today, especially those who have large investments in C. Traditionally, OOP languages have suffered from the attitude that you should abandon everything you know and start from scratch with a new set of concepts and a new syntax, arguing that it's better in the long run to lose all the old baggage that comes with procedural languages. This may be true, in the long run. But in the short run, a lot of that baggage was valuable. The most valuable elements may not be the existing code base (which, given adequate tools, could be translated), but instead the existing *mind base*. If you're a functioning C programmer and must drop everything you know about C in order to adopt a new language, you immediately become much less productive for many months, until your mind fits around the new paradigm. Whereas if you can leverage off of your existing C knowledge and expand on it, you can continue to be productive with what you already know while moving into the world of object-oriented programming. As everyone has his or her own mental model of programming, this move is messy enough as it is without the added expense of starting with a new language model from square one. So the reason for the success of C++, in a nutshell, is economic: It still costs to move to OOP, but C++ may cost less.

The goal of C++ is improved productivity. This productivity comes in many ways, but the language is designed to aid you as much as possible, while hindering you as little as possible with arbitrary rules or any requirement that you use a particular set of features. C++ is designed to be practical; C++ language design decisions were based on providing the maximum benefits to the programmer (at least, from the world view of C).

A better C

You get an instant win even if you continue to write C code because C++ has closed many holes in the C language and provides better type checking and compile-time analysis. You're forced to declare functions so that the compiler can check their use. The need for the preprocessor has virtually been eliminated for value substitution and macros, which removes a set of difficult-to-find bugs. C++ has a feature called *references* that allows more convenient handling of addresses for function arguments and return values. The handling of names is improved through a feature called *function overloading*, which allows you to use the same name for different functions. A feature called *namespaces* also improves the control of names. There are numerous smaller features that improve the safety of C.

You're already on the learning curve

The problem with learning a new language is productivity. No company can afford to suddenly lose a productive software engineer because he or she is learning a new language. C++ is an extension to C, not a complete new syntax and programming model. It allows you to continue creating useful code, applying the features gradually as you learn and understand them. This may be one of the most important reasons for the success of C++.

In addition, all of your existing C code is still viable in C++, but because the C++ compiler is pickier, you'll often find hidden C errors when recompiling the code in C++.

Efficiency

Sometimes it is appropriate to trade execution speed for programmer productivity. A financial model, for example, may be useful for only a short period of time, so it's more important to create the model rapidly than to execute it rapidly. However, most applications require some degree of efficiency, so C++ always errs on the side of greater efficiency. Because C programmers tend to be very efficiency-conscious, this is also a way to ensure that they won't be able to argue that the language is too fat and slow. A number of features in C++ are intended to allow you to tune for performance when the generated code isn't efficient enough.

Not only do you have the same low-level control as in C (and the ability to directly write assembly language within a C++ program), but anecdotal evidence suggests that the program speed for an object-oriented C++ program tends to be within ±10% of a program written in C, and often much closer. The design produced for an OOP program may actually be more efficient than the C counterpart.

Systems are easier to express and understand

Classes designed to fit the problem tend to express it better. This means that when you write the code, you're describing your solution in the terms of the problem space ("Put the grommet in the bin") rather than the terms of the computer, which is the solution space ("Set the bit in the chip that means that the relay will close"). You deal with higher-level concepts and can do much more with a single line of code.

The other benefit of this ease of expression is maintenance, which (if reports can be believed) takes a huge portion of the cost over a program's lifetime. If a program is easier to understand, then it's easier to maintain. This can also reduce the cost of creating and maintaining the documentation.

Maximal leverage with libraries

The fastest way to create a program is to use code that's already written: a library. A major goal in C++ is to make library use easier. This is accomplished by casting libraries into new data types (classes), so that bringing in a library means adding new types to the language. Because the C++ compiler takes care of how the library is used – guaranteeing

proper initialization and cleanup, and ensuring that functions are called properly – you can focus on what you want the library to do, not how you have to do it.

Because names can be sequestered to portions of your program via C++ namespaces, you can use as many libraries as you want without the kinds of name clashes you'd run into with C.

Source-code reuse with templates

There is a significant class of types that require source-code modification in order to reuse them effectively. The *template* feature in C++ performs the source code modification automatically, making it an especially powerful tool for reusing library code. A type that you design using templates will work effortlessly with many other types. Templates are especially nice because they hide the complexity of this kind of code reuse from the client programmer.

Error handling

Error handling in C is a notorious problem, and one that is often ignored – finger-crossing is usually involved. If you're building a large, complex program, there's nothing worse than having an error buried somewhere with no clue as to where it came from. C++ *exception handling* (introduced in this Volume, and fully covered in Volume 2, which is downloadable from *www.BruceEckel.com*) is a way to guarantee that an error is noticed and that something happens as a result.

Programming in the large

Many traditional languages have built-in limitations to program size and complexity. BASIC, for example, can be great for pulling together quick solutions for certain classes of problems, but if the program gets more than a few pages long or ventures out of the normal problem domain of that language, it's like trying to swim through an ever-more viscous fluid. C, too, has these limitations. For example, when a program gets beyond perhaps 50,000 lines of code, name collisions start to become a problem – effectively, you run out of function and variable names. Another particularly bad problem is the little holes in the C language – errors buried in a large program can be extremely difficult to find.

There's no clear line that tells you when your language is failing you, and even if there were, you'd ignore it. You don't say, "My BASIC program just got too big; I'll have to rewrite it in C!" Instead, you try to shoehorn a few more lines in to add that one new feature. So the extra costs come creeping up on you.

C++ is designed to aid *programming in the large*, that is, to erase those creeping-complexity boundaries between a small program and a large one. You certainly don't need to use OOP, templates, namespaces, and exception handling when you're writing a hello-world style utility program, but those features are there when you need them. And

the compiler is aggressive about ferreting out bug-producing errors for small and large programs alike.