Chapter 0: What is C++?

C++ is a general purpose programming language with a bias towards systems programming that

- *is a better C.*
- *supports data abstraction.*
- *supports object-oriented programming.*
- *supports generic programming*

– Bjarne Stroustrup, inventor of C++ [Str09.2]

Every programming language has its own distinct flavor influenced by its history and design. Before seriously studying a programming language, it's important to learn *why* the language exists and what its objectives are. This chapter covers a quick history of C++, along with some of its design principles.

An Abbreviated History of C++*

The story of C++ begins with Bjarne Stroustrup, a Danish computer scientist working toward his PhD at Cambridge University. Stroustrup's research focus was *distributed systems*, software systems split across several computers that communicated over a network to solve a problem. At one point during his research, Stroustrup came up with a particularly clever idea for a distributed system. Because designing distributed systems is an enormously complicated endeavor, Stroustrup decided to test out his idea by writing a simulation program, which is a significantly simpler task. Stroustrup chose to write this simulation program in a language called Simula, one of the earliest object-oriented programming languages. As Stroustrup recalled, initially, Simula seemed like the perfect tool for the job:

> It was a pleasure to write that simulator. The features of Simula were almost ideal for the purpose, and I was particularly impressed by the way the concepts of the language helped me think about the problems in my application. The class concept allowed me to map my application concepts into the language constructs in a direct way that made my code more readable than I had seen in any other language...
>
> I had used Simula before... but was very pleasantly surprised by the way the mechanisms of the Simula language became increasingly helpful as the size of the program increased. [Str94]

In Simula, it was possible to model a *physical* computer using a computer *object* and a *physical* network using a network *object*, and the way that physical computers sent packets over physical networks corresponded to the way computer objects sent and received messages from network objects. But while Simula made it easier for Stroustrup to develop the simulator, the resulting program was so slow that it failed to produce any meaningful results. This was not the fault of Stroustrup's implementation, but of the language Simula itself. Simula was bloated and language features Stroustrup didn't use in his program were crippling the simulator's efficiency. For example, Stroustrup found that eighty percent of his program time was being spent on garbage collection despite the fact that the simulation didn't create any garbage. [Str94] In other words, while Simula had *decreased* the time required to build the simulator, it dramatically *increased* the time required for the simulator to execute.

Stroustrup realized that his Simula-based simulator was going nowhere. To continue his research, Stroustrup scrapped his Simula implementation and rewrote the program in a language he knew ran quickly and efficiently: BCPL. BCPL has since gone the way of the dodo, but at the time was a widely used, low-level systems programming language. Stroustrup later recalled that writing the simulator in BCPL was “horrible.” [Str94] As a

* This section is based on information from *The Design and Evolution of C++* by Bjarne Stroustrup.

low-level language, BCPL lacked objects and to represent computers and networks Stroustrup had to manually lay out and manipulate the proper bits and bytes. However, BCPL programs were far more efficient than their Simula counterparts, and Stroustrup's updated simulator worked marvelously.

Stroustrup's experiences with the distributed systems simulator impressed upon him the need for a more suitable tool for constructing large software systems. Stroustrup sought a hybridization of the best features of Simula and BCPL – a language with both high-level constructs and low-level runtime efficiency. After receiving his PhD, Stroustrup accepted a position at Bell Laboratories and began to create such a language. Settling on C as a base language, Stroustrup incorporated high-level constructs in the style of Simula while still maintaining C's underlying efficiency. After several revisions, *C with Classes*, as his language was known, accumulated other high-level features and was officially renamed C++. C++ was an overnight success and spread rapidly into the programming community; for many years the number of C++ programmers was doubling every seven months. By 2007, there were over three million C++ programmers worldwide, and despite competition from other languages like Java and Python the number of C++ programmers is still increasing. [Str09] What began as Stroustrup's project at Bell Laboratories became an ISO-standardized programming language found in a variety of applications.

C++ as a Language

When confronted with a new idea or concept, it's often enlightening to do a quick Wikipedia search to see what others have to say on the subject. If you look up C++ this way, one of the first sentences you'll read (at least, at the time of this writing) will tell you that C++ is a general-purpose, compiled, statically-typed, multiparadigm, mid-level programming language. If you are just learning C++, this description may seem utterly mystifying. However, this sentence very aptly captures much of the spirit of C++, and so before continuing our descent into the realm of C++ let's take a few minutes to go over exactly what this definition entails.

C++ is a General-Purpose Programming Language

Programming languages can be broadly categorized into two classes – domain-specific programming languages and general-purpose programming languages. A language is *domain-specific* if it is designed to solve a certain class of problems in a particular field. For example, the MATLAB programming language is a domain-specific language designed for numerical and mathematical computing, and so has concise and elegant support for matrix and vector operations. Domain-specific languages tend to be extremely easy to use, particularly because these languages let programmers express common operations concisely and elegantly because the language has been designed with them in mind. As an example, in MATLAB it is possible to solve a linear system of equations using the simple syntax `x = A\b`. The equivalent C++ or Java code would be significantly more complex. However, because domain-specific languages are optimized on a particular class of problems, it can be difficult if not impossible to adapt those languages into other problem domains. This has to do with the fact that domain-specific languages are custom-tailored to the problems they solve, and consequently lack the vocabulary or syntactic richness to express structures beyond their narrow scope. This is best illustrated by analogy – an extraordinary mathematician with years of training would probably have great difficulty holding a technical discussion on winemaking with the world's expert oenologist simply because the vocabularies of mathematics and winemaking are entirely different. It might be possible to explain viticulture to the mathematician using terms from differential topology or matrix theory, but this would clearly be a misguided effort.

Contrasting with domain-specific languages are *general-purpose* languages which, as their name suggests, are designed to tackle all categories of problems, not just one particular class. This means that general-purpose languages are more readily adapted to different scenarios and situations, but may have a harder time describing some of the fundamental concepts of those domains than a language crafted specifically for that purpose. For example, an American learning German as a second language may be fluent enough in that language to converse with strangers and to handle day-to-day life, but might have quite an experience trying to hold a technical conversation with industry specialists. This is not to say, of course, that the American would not be able to comprehend the ideas that the specialist was putting forth, but rather that any discussion the two would have would re-

quire the specialist to define her terms as the conversation unfolded, rather than taking their definitions for granted at the start.

C++ is a general-purpose programming language, which means that it is robust enough to adapt to handle all sorts of problems without providing special tools that simplify tasks in any one area. This is a trade-off, of course. Because C++ is general-purpose, it will not magically provide you a means for solving a particular problem; you will have to think through a design for your programs in order for them to work correctly. But because C++ is general-purpose, you will be hard-pressed to find a challenge for which C++ is a poor choice for the solution. Moreover, because C++ is a general-purpose language, once you have learned the structures and techniques of C++, you can apply your knowledge to any problem domain without having to learn new syntax or structures designed for that domain.

C++ is a Compiled Language

The programs that actually execute on a computer are written in machine language, an extremely low-level and hardware-specific language that encodes individual instructions for the computer's CPU. Machine languages are indecipherable even to most working programmers because these languages are designed to be read by computer hardware rather than humans. Consequently, programmers write programs in programming languages, which are designed to be read by humans. In order to execute a program written in a programming language, that program must somehow be converted from its source code representation into equivalent machine code for execution. How this transformation is performed is not set in stone, and in general there are two major approaches to converting source code to machine code. The first of these is to *interpret* the program. In *interpreted languages*,a special program called the *interpreter* takes in the program's source code and translates the program as it is being executed. Whenever the program needs to execute a new piece of code, the interpreter reads in the next bit of the source code, converts it into equivalent machine code, then executes the result. This means that if the same interpreted program is run several times, the interpreter will translate the program anew every time. The other option is to *compile* the program. In a *compiled language*, before running the program, the programmer executes a special program called the *compiler* on the source code which translates the entire program into machine code. This means that no matter how many times the resulting program is run, the compiler is only invoked once. In general, interpreted languages tend to run more slowly than compiled languages because the interpreter must translate the program as it is being executed, whereas the translation work has already been done in the case of compiled languages. Because C++ places a premium on efficiency, C++ is a compiled language. While C++ interpreters do exist, they are almost exclusively for research purposes and rarely (if at all) used in professional settings.

What does all of this mean for you as a C++ programmer? That is, why does it matter whether C++ is compiled or interpreted? A great deal, it turns out; this will be elaborated upon in the next segment on static type checking. However, one way that you will notice immediately is that you will have to compile your programs every time you make a change to the source code that you want to test out. When working on very large software projects (on the order of millions to hundreds of millions of lines of code), it is not uncommon for a recompilation to take hours to complete, meaning that it is difficult to test out lots of minor changes to a C++ program. After all, if every change takes three minutes to test, then the number of possible changes you can make to a program in hopes of eliminating a bug or extending functionality can be greatly limited. On the other hand, though, because C++ is compiled, once you have your resulting program it will tend to run much, *much* faster than programs written in other languages. Moreover, you don't need to distribute an interpreter for your program in addition to the source – because C++ programs compile down directly to the machine code, you can just ship an executable file to whoever wants to run your program and they should be able to run it without any hassle.

C++ is a Statically-Typed Language

One of the single most important aspects of C++ is that it is a statically-typed language. If you want to manipulate data in a C++ program, you must specify in advance what the *type* of that data is (for example, whether it's an integer, a real number, English text, a jet engine, etc.). Moreover, this type is set in stone and cannot change elsewhere in the source code. This means that if you say that an object is a coffee mug, you cannot treat it as a stapler someplace else.

At first this might seem silly – *of course* you shouldn't be able to convert a coffee mug into a stapler or a ball of twine into a jet engine; those are entirely different entities! You are completely correct about this. Any program that tries to treat a coffee mug as though it is a stapler is bound to run into trouble because a coffee mug *isn't* a stapler. The reason that static typing is important is that these sorts of errors are caught at *compile-time* instead of at *runtime*. This means that if you write a program that tries to make this sort of mistake, the program won't compile and you won't even have an executable containing a mistake to run. If you write a C++ program that tries to treat a coffee mug like a stapler, the compiler will give you an error and you will need to fix the problem before you can test out the program. This is an extremely powerful feature of compiled languages and will dramatically reduce the number of runtime errors that your programs encounter. As you will see later in this book, this also enables you to have the compiler verify that complex relationships hold in your code and can conclude that if the program compiles, your code does not contain certain classes of mistakes.

C++ is a Multi-Paradigm Language

C++ began as a hybrid of high- and low-level languages but has since evolved into a distinctive language with its own idioms and constructs. Many programmers treat C++ as little more than an object-oriented C, but this view obscures much of the magic of C++. C++ is a *multiparadigm* programming language, meaning that it supports several different programming styles. C++ supports *imperative* programming in the style of C, meaning that you can treat C++ as an upgraded C. C++ supports *object-oriented* programming, so you can construct elaborate class hierarchies that hide complexity behind simple interfaces. C++ supports *generic* programming, allowing you to write code reusable in a large number of contexts. Finally, C++ supports a limited form of *higher-order* programming, allowing you to write functions that construct and manipulate other functions at runtime.

C++ being a multiparadigm language is both a blessing and a curse. It is a blessing in that C++ will let you write code in the style that you feel is most appropriate for a given problem, rather than rigidly locking you into a particular framework. It is also a blessing in that you can mix and match styles to create programs that are precisely suited for the task at hand. It is a curse, however, in that multiparadigm languages are necessarily more complex than single-paradigm languages and consequently C++ is more difficult to pick up than other languages. Moreover, the interplay among all of these paradigms is complex, and you will need to learn the subtle but important interactions that occur at the interface between these paradigms.

This book is organized so that it covers a mixture of all of the aforementioned paradigms one after another, and ideally you will be comfortable working in each by the time you've finished reading.

C++ is a Mid-Level Language

Computer programs ultimately must execute on computers. Although computers are capable of executing programs which perform complex abstract reasoning, the computers themselves understand only the small set of commands necessary to manipulate bits and bytes and to perform simple arithmetic. Low-level languages are languages like C and assembly language that provide minimal structure over the actual machine and expose many details about the inner workings of the computer. To contrast, high-level languages are languages that abstract away from the particulars of the machine and let you write programs independently of the computer's idiosyncrasies. As mentioned earlier, low-level languages make it hard to represent complex program structure, while high-level languages often are too abstract to operate efficiently on a computer.

C++ is a rare language in that it combines the low-level efficiency and machine access of C with high-level constructs like those found in Java. This means that it is possible to write C++ programs with the strengths of both approaches. It is not uncommon to find C++ programs that model complex systems using object-oriented techniques (high level) while taking advantage of specific hardware to accelerate that simulation (low-level). One way to think about the power afforded by C++ is to recognize that C++ is a language that provides a set of abstractions that let you intuitively design large software systems, but which lets you break those abstractions when the need to optimize becomes important. We will see some ways to accomplish this later in this book.

Design Philosophy

C++ is a comparatively old language; its first release was in 1985. Since then numerous other programming languages have sprung up – Java, Python, C#, and Javascript, to name a few. How exactly has C++ survived so long when others have failed? C++ may be useful and versatile, but so were BCPL and Simula, neither of which are in widespread use today.

One of the main reasons that C++ is still in use (and evolving) today has been its core guiding principles. Stroustrup has maintained an active interest in C++ since its inception and has steadfastly adhered to a particular design philosophy. Here is a sampling of the design points, as articulated in Stroustrup's *The Design and Evolution of C++*.

- **C++'s evolution must be driven by real problems**. When existing programming styles prove insufficient for modern challenges, C++ adapts. For example, the introduction of exception handling provided a much-needed system for error recovery, and abstract classes allowed programmers to define interfaces more naturally.

- **Don't try to force people**. C++ supports multiple programming styles. You can write code similar to that found in pure C, design class hierarchies as you would in Java, or develop software somewhere in between the two. C++ respects and trusts you as a programmer, allowing you to write the style of code you find most suitable to the task at hand rather than rigidly locking you into a single pattern.

- **Always provide a transition path**. C++ is designed such that the programming principles and techniques developed at any point in its history are still applicable. With few exceptions, C++ code written ten or twenty years ago should still compile and run on modern C++ compilers. Moreover, C++ is designed to be mostly backwards-compatible with C, meaning that veteran C coders can quickly get up to speed with C++.

The Goal of C++

There is one quote from Stroustrup ([Str94]) I believe best sums up C++:

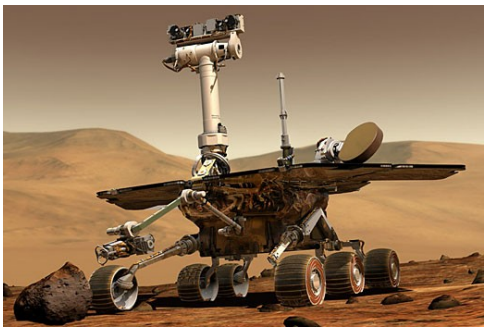
*C++ makes programming **more enjoyable** for **serious programmers**.*

What exactly does this mean? Let's begin with what constitutes a *serious programmer.* Rigidly defining "serious programmer" is difficult, so instead I'll list some of the programs and projects written in C++ and leave it as an exercise to the reader to infer a proper definition. For example, you'll find C++ in:

Mozilla Firefox. The core infrastructure underlying all Mozilla projects is written predominantly in C++. While much of the code for Firefox is written in Javascript and XUL, these languages are executed by interpreters written in C++.

The WebKit layout engine used by Safari and Google Chrome is also written in C++. Although it's closed-source, I suspect that Internet Explorer is also written in C++. If you're browsing the web, you're seeing C++ in action.

Java HotSpot. The widespread success of Java is in part due to *HotSpot*, Sun's implementation of the Java Virtual Machine. HotSpot supports just-in-time compilation and optimization and is a beautifully engineered piece of software. It's also written in C++. The next time that someone engages you in a debate about the relative merits of C++ and Java, you can mention that if not for a well-architected C++ program Java would not be a competitive language.



NASA / JPL. The rovers currently exploring the surface of Mars have their autonomous driving systems written in C++. *C++ is on Mars!*

C++ makes programming ***more enjoyable*** *for serious programmers*. Not only does C++ power all of the above applications, it powers them *in style*. You can program with high-level constructs yet enjoy the runtime efficiency of a low-level language like C. You can choose the programming style that's right for you and work in a language that trusts and respects your expertise. You can write code once that you will reuse time and time again. This is what C++ is all about, and the purpose of this book is to get you up to speed on the mechanics, style, and just plain excitement of C++.

With that said, let's dive into C++. Our journey begins!