



CHAPTER 1

The Story of C++

C++ is the single most important language that any programmer can learn. This is a strong statement, but it is not an exaggeration. C++ is the center of gravity around which all of modern programming revolves. Its syntax and design philosophy define the essence of object-oriented programming. Moreover, C++ charts the course for future language development. For example, both Java and C# are directly descended from C++. C++ is also the universal language of programming; it is the language in which programmers share ideas with one another. To be a professional programmer today implies competency in C++. It is that fundamental and that important. C++ is the gateway to all of modern programming.

Before beginning your study of C++, it is important for you to know how C++ fits into the historical context of computer languages. Understanding the forces that drove its creation, the design philosophy it represents, and the legacy that it inherits makes it easier to appreciate the many innovative and unique features of C++. With this in mind, this chapter presents a brief history of the C++ programming language, its origins, its relationship to its predecessor (C), its uses, and the programming philosophies that it supports. It also puts C++ into perspective relative to other programming languages.

The Origins of C++

The story of C++ begins with C. The reason for this is simple: C++ is built upon the foundation of C. In fact, C++ is a superset of C. (Indeed, all C++ compilers can also be used to compile C programs!) Specifically, C++ is an expanded and enhanced version of C that embodies the philosophy of object-oriented programming (which is described later in this chapter). C++ also includes several other improvements to the C language, including an extended set of library routines. However, much of the spirit and flavor of C++ is inherited directly from C. To fully understand and appreciate C++, you need to understand the “how and why” behind C.

The Creation of C

The C language shook the computer world. Its impact should not be underestimated because it fundamentally changed the way programming was approached and thought about. C is considered by many to be the first modern “programmer’s language.” Prior to the invention of C, computer languages were generally designed either as academic exercises or by bureaucratic committees. C is different. C was designed, implemented, and developed by real, working programmers, and it reflected the way they approached the job of programming. Its features were honed, tested, thought about, and rethought by the people who actually used the language. The result of this process was a language that programmers liked to use. Indeed, C quickly attracted many followers who had a near-religious zeal for it, and it found wide and rapid acceptance in the programmer community. In short, C is a language designed by and for programmers.

C was invented and first implemented by Dennis Ritchie on a DEC PDP-11 using the UNIX operating system. C is the result of a development process that started with an older language called BCPL, which was developed by Martin Richards. BCPL influenced a language called B, invented by Ken Thompson, which led to the development of C in the 1970s.

For many years, the de facto standard for C was the one supplied with the Unix operating system and described in *The C Programming Language*, by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). However, because no formal standard existed, there were discrepancies between different implementations of C. To alter this situation, a committee was established in the beginning of the summer of 1983 to work on the creation of an ANSI (American National Standards Institute) standard that would define—once and for all—the C language. The final version of the standard was adopted in December 1989, the first copies of which became available in early 1990. This version of C is commonly referred to as *C89*, and it is the foundation upon which C++ is built.



NOTE: The C standard was updated in 1999 and this version of C is usually referred to as *C99*. This version contains some new features, including a few borrowed from C++, but, overall, it is compatible with the original C89 standard. At the time of this writing, no widely available compiler supports C99 and it is still C89 that defines what is commonly thought of as the C language. Furthermore, it is C89 that is the basis for C++. It is possible that a future standard for C++ will include the features added by C99, but they are not part of C++ at this time.

It may seem hard to understand at first, but C is often called a “middle-level” computer language. As it is applied to C, middle-level does not have a negative connotation; it does not mean that C is less powerful, harder to use, or less developed than a “high-level” language, or that it is as difficult to use as assembly language. (*Assembly language*, or *assembler*, as it is often called, is simply a symbolic representation of the actual machine code that a computer can execute.) C is thought of as a middle-level language because it combines elements of high-level languages, such as Pascal, Modula-2, or Visual Basic, with the functionality of assembler.

From a theoretical point of view, a *high-level language* attempts to give the programmer everything he or she could possibly want, already built into the language. A *low-level language* provides nothing other than access to the actual machine instructions. A *middle-level language* gives the programmer a concise set of tools and allows the programmer to develop higher-level constructs on his or her own. A middle-level language offers the programmer built-in power, coupled with flexibility.

Being a middle-level language, C allows you to manipulate bits, bytes, and addresses—the basic elements with which a computer functions. Thus, C does not attempt to buffer the hardware of the machine from your program to any significant extent. For example, the size of an integer in C is directly related to the word size of the CPU. In most high-level languages there are built-in statements for reading and writing disk files. In C, all of these procedures are performed by calls to library routines and not by keywords defined by the language. This approach increases C’s flexibility.

C allows—indeed, needs—the programmer to define routines for performing high-level operations. These routines are called *functions*, and they are very important to the C language. In fact, functions are the building blocks of both C and C++. You can easily tailor a library of functions to perform various tasks that are used by your program. In this sense, you can personalize C to fit your needs.

There is another aspect of C that you must understand, because it is also important to C++: C is a structured language. The most distinguishing feature of a structured language is that it uses blocks. A *block* is a set of statements that are logically connected. For example, imagine an IF statement that, if successful, will execute five discrete statements. If these statements can be grouped together and referenced as an indivisible unit, then they form a block.

A structured language supports the concept of subroutines with local variables. A *local variable* is simply a variable that is known only to the subroutine in which it is defined. A structured language also supports several loop constructs, such as **while**, **do-while**, and **for**. The use of the **goto** statement, however, is either prohibited or discouraged, and is not the common form of program control in the same way that it is in traditional BASIC or FORTRAN. A structured language allows you to indent statements and does not require a strict field concept (as did early versions of FORTRAN).

Finally, and perhaps most importantly, C is a language that stays out of the way. The underlying philosophy of C is that the programmer, not the language, is in charge. Therefore, C will let you do virtually anything that you want, even if what you tell it to do is unorthodox, highly unusual, or suspicious. C gives you nearly complete control over the machine. Of course, with this power comes considerable responsibility, which you, the programmer, must shoulder.

Understanding the Need for C++

Given the preceding discussion of C, you might be wondering why C++ was invented. Since C is a successful and useful computer programming language, why was there a need for something else? The answer is complexity. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand this correlation, consider the following.

Approaches to programming have changed dramatically since the invention of the computer. The primary reason for change has been to accommodate the increasing complexity of programs. For example, when computers were first invented, programming was done by toggling in the binary machine instructions using the computer's front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that programmers could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were developed to give programmers more tools with which to handle complexity.

The first widespread language was, of course, FORTRAN. While FORTRAN was a very impressive first step, it is hardly a language that encourages clear, easy-to-understand programs. The 1960s gave birth to structured programming. This is the method of programming supported by languages such as C. With structured languages, it was, for the first time, possible to write moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. By the late 1970s, many projects were near or at this point. To solve this problem, a new way to program began to emerge. This method is called *object-oriented programming* (OOP for short). Using OOP,

a programmer could handle larger programs. The trouble was that C did not support object-oriented programming. The desire for an object-oriented version of C ultimately led to the creation of C++.

In the final analysis, although C is one of the most liked and widely used professional programming languages in the world, there comes a time when its ability to handle complexity reaches its limit. The purpose of C++ is to allow this barrier to be broken and to help the programmer comprehend and manage larger, more complex programs.

C++ Is Born

In response to the need to manage greater complexity, C++ was born. It was invented by Bjarne Stroustrup in 1979 at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language “C with Classes.” However, in 1983 the name was changed to C++.

C++ contains the entire C language. As stated earlier, C is the foundation upon which C++ is built. C++ includes all of C’s features, attributes, and benefits. It also adheres to C’s philosophy that the programmer, not the language, is in charge. At this point, it is critical to understand that the invention of C++ was not an attempt to create a new programming language. Instead, it was an enhancement to an already highly successful language.

Most of the additions that Stroustrup made to C were designed to support object-oriented programming. In essence, C++ is the object-oriented version of C. By building upon the foundation of C, Stroustrup provided a smooth migration path to OOP. Instead of having to learn an entirely new language, a C programmer needed to learn only a few new features to reap the benefits of the object-oriented methodology.

But C is not the only language that influenced C++. Stroustrup states that some of its object-oriented features were inspired by another object-oriented language called Simula67. Therefore, C++ represents the blending of two powerful programming methods.

When creating C++, Stroustrup knew that it was important to maintain the original spirit of C, including its efficiency, flexibility, and philosophy, while at the same time adding support for object-oriented programming. Happily, his goal was accomplished. C++ still provides the programmer with the freedom and control of C, coupled with the power of objects.

Although C++ was initially designed to aid in the management of very large programs, it is in no way limited to this use. In fact, the object-oriented attributes of C++ can be effectively applied to virtually any programming task. It is not uncommon to see C++ used for projects such as compilers, editors, programmer tools, games, and networking programs. Because C++ shares C’s efficiency, much high-performance systems software is constructed using C++. Also, C++ is frequently the language of choice for Windows programming.

One important point to remember is this: Because C++ is a superset of C, once you can program in C++, you can also program in C! Thus, you will actually be learning two programming languages at the same time, with the same effort that you would use to learn only one.

The Evolution of C++

Since C++ was first invented, it has undergone three major revisions, with each revision adding to and altering the language. The first revision was in 1985 and the second occurred in 1990. The third revision occurred during the C++ standardization process. In the early 1990s, work began on a standard for C++. Towards that end, a joint ANSI and ISO (International Standards Organization) standardization committee was formed. The first draft of the proposed standard was created on January 25, 1994. In that draft, the ANSI/ISO C++ committee (of which I was a member) kept the features first defined by Stroustrup and added some new ones as well. But, in general, this initial draft reflected the state of C++ at the time.

Soon after the completion of the first draft of the C++ standard, an event occurred that caused the standard to expand greatly: the creation of the Standard Template Library (STL) by Alexander Stepanov. As you will learn, the STL is a set of generic routines that you can use to manipulate data. It is both powerful and elegant. But the STL is also quite large. Subsequent to the first draft, the committee voted to include the STL in the specification for C++. The addition of the STL expanded the scope of C++ well beyond its original definition. While important, the inclusion of the STL, among other things, slowed the standardization of C++.

It is fair to say that the standardization of C++ took far longer than any one had expected when it began. In the process, many new features were added to the language and many small changes were made. In fact, the version of C++ defined by the C++ committee is much larger and more complex than Stroustrup's original design. The final draft was passed out of committee on November 14, 1997, and an ANSI/ISO standard for C++ became a reality in 1998. This specification for C++ is commonly referred to as *Standard C++*.

The material in this book describes Standard C++. This is the version of C++ supported by all mainstream C++ compilers, including Microsoft's Visual C++ and Borland's C++ Builder. Therefore, the code and information in this book is fully applicable to all modern C++ environments.

What Is Object-Oriented Programming?

Since object-oriented programming was fundamental to the development of C++, it is important to define precisely what object-oriented programming is. Object-oriented programming has taken the best ideas of structured programming and has combined them with several powerful concepts that allow you to organize your programs more effectively. In general, when programming in an object-oriented fashion, you decompose a problem into its constituent parts. Each component becomes a self-contained object that contains its own instructions and data related to that object. Through this process, complexity is reduced and you can manage larger programs.

All object-oriented programming languages have three things in common: encapsulation, polymorphism, and inheritance. Although we will examine these concepts in detail later in this book, let's take a brief look at them now.

Encapsulation

As you probably know, all programs are composed of two fundamental elements: program statements (code) and data. *Code* is that part of a program that performs actions, and *data* is the information affected by those actions. *Encapsulation* is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse.

In an object-oriented language, code and data may be bound together in such a way that a self-contained *black box* is created. Within the box are all necessary data and code. When code and data are linked together in this fashion, an object is created. In other words, an *object* is the device that supports encapsulation.

Within an object, the code, data, or both may be private to that object or public. *Private* code or data is known to, and accessible only by, another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is *public*, other parts of your program may access it, even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

Polymorphism

Polymorphism (from the Greek, meaning “many forms”) is the quality that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. A simple example of polymorphism is found in the steering wheel of an automobile. The steering wheel (i.e., the interface) is the same no matter what type of actual steering mechanism is used. That is, the steering wheel works the same whether your car has manual steering, power steering, or rack-and-pinion steering. Therefore, once you know how to operate the steering wheel, you can drive any type of car. The same principle can also apply to programming. For example, consider a stack (which is a first-in, last-out list). You might have a program that requires three different types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. In this case, the algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, calling each set by a different name, with each set having its own interface. However, because of polymorphism, in C++ you can create one general set of stack routines (one interface) that works for all three specific situations. This way, once you know how to use one stack, you can use them all.

More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. Polymorphism helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler’s job to select the *specific action* (i.e., method) as it applies to each situation. You, the programmer, don’t need to do this selection manually. You need only remember and utilize the general interface.

The first object-oriented programming languages were interpreters, so polymorphism was, of course, supported at run time. However, C++ is a compiled language. Therefore, in C++, both run-time and compile-time polymorphism are supported.

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. The reason this is important is that it supports the concept of hierarchical classification. If you think about it, most knowledge is made manageable by hierarchical (i.e., top-down) classifications. For example, a Red Delicious apple is part of the classification *apple*, which in turn is part of the *fruit* class, which is under the larger class *food*. That is, the *food* class possesses certain qualities (edible, nutritious, etc.) that also apply, logically, to its *fruit* subclass. In addition to these qualities, the *fruit* class has specific characteristics (juicy, sweet, etc.) that distinguish it from other food. The *apple* class defines those qualities specific to an apple (grows on trees, not tropical, etc.). A Red Delicious apple would, in turn, inherit all the qualities of all preceding classes, and would define only those qualities that make it unique.

Without the use of hierarchies, each object would have to explicitly define all of its characteristics. However, using inheritance, an object needs to define only those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

C++ Implements OOP

As you will see as you progress through this book, many of the features of C++ exist to provide support for encapsulation, polymorphism, and inheritance. Remember, however, that you can use C++ to write any type of program, using any type of approach. The fact that C++ supports object-oriented programming does not mean that you can only write object-oriented programs. As with its predecessor, C, one of C++'s strongest advantages is its flexibility.

How C++ Relates to Java and C#

As most readers will know, there are two other computer languages that are having a strong impact on programming: Java and C#. Java was developed by Sun Microsystems and C# was created by Microsoft. Because there is sometimes confusion about how these two languages relate to C++, a brief discussion of their relationship is in order.

C++ is the parent for both Java and C#. Although Java and C# added, removed, and modified various features, in total the syntax for all three languages is nearly identical. Furthermore, the object model used by C++ is similar to the ones used by Java and C#. Finally, the overall “look and feel” of these languages is very similar. This means that once you know C++, you can easily learn Java or C#. This is one reason that Java and C# borrowed C++'s syntax and object model; it facilitated their rapid adoption by legions of experienced C++ programmers. The reverse case is also true. If you know Java or C#, learning C++ is easy.

The main difference between C++, Java, and C# is the type of computing environment for which each is designed. C++ was created to produce high-performance programs for a specific type of CPU and operating system. For example, if you want to write a high-performance program that runs on an Intel Pentium under the Windows operating system, then C++ is the best language to use.

Java and C# were developed in response to the unique programming needs of the highly distributed networked environment that typifies much of contemporary computing. Java was designed to enable the creation of cross-platform portable code for the Internet. Using Java, it is possible to write a program that runs in a wide variety of environments, on a wide range of operating systems and CPUs. Thus, a Java program can move about freely on the Internet. C# was designed for Microsoft's .NET Framework, which supports mixed-language, component-based code that works in a networked environment.

Although both Java and C# enable the creation of portable code that works in a highly distributed environment, the price one pays for this portability is efficiency. Java programs execute slower than do C++ programs. The same is true for C#. Thus, if you want to create high-performance software, use C++. If you need to create highly portable software, use Java or C#.

One final point: C++, Java, and C# are designed to solve different sets of problems. It is not an issue of which language is best in and of itself. Rather, it is a question of which language is right for the job at hand.