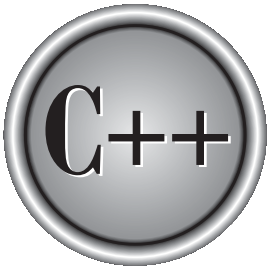
The Complete Reference



Chapter 1

An Overview of C

To understand C++ is to understand the forces that drove its creation, the ideas that shaped it, and the legacy it inherits. Thus, the story of C++ begins with C. This chapter presents an overview of the C programming language, its origins, its uses, and its underlying philosophy. Because C++ is built upon C, this chapter provides an important historical perspective on the roots of C++. Much of what "makes C++, C++" had its genesis in the C language.

The Origins and History of C

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

For many years, the de facto standard for C was the version supplied with the UNIX operating system. It was first described in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Englewood Cliffs, N.J.: Prentice-Hall, 1978). In the summer of 1983 a committee was established to create an ANSI (American National Standards Institute) standard that would define the C language. The standardization process took six years (much longer than anyone reasonably expected at the time).

The ANSI C standard was finally adopted in December of 1989, with the first copies becoming available in early 1990. The standard was also adopted by ISO (International Standards Organization) and the resulting standard was typically referred to as ANSI/ISO Standard C. In 1995, Amendment 1 to the C standard was adopted, which, among other things, added several new library functions. The 1989 standard for C, along with Amendment 1, became the *base document* for Standard C++, defining the *C subset* of C++. The version of C defined by the 1989 standard is commonly referred to as C89.

After 1989, C++ took center stage, and during the 1990s the development of a standard for C++ consumed most programmers' attention, with a standard for C++ being adopted by the end of 1998. However, work on C continued along quietly. The end result was the 1999 standard for C, usually referred to as C99. In general, C99 retained nearly all of the features of C89 and did not alter the main aspects of the language. Thus, the C language described by C99 is essentially the same as the one described by C89. The C99 standardization committee focused on two main areas: the addition of several numeric libraries and the development of some special-use, but highly innovative, new features, such as variable- length arrays and the **restrict** pointer qualifier. In a few cases, features originally from C++, such as single-line comments, were also incorporated into C99. Because the standard for C++ was finalized before C99 was created, none of the C99 innovations are found in Standard C++.

C89 vs. C99

Although the innovations in C99 are important from a computer science point of view, they are currently of little practical consequence because, at the time of this writing, no widely-used compiler implements C99. Rather, it is C89 that defines the version of C

that most programmers think of as "C" and that all mainstream compilers recognize. Furthermore, it is C89 that forms the C subset of C++. Although several of the new features added by C99 will eventually find their way into the next standard for C++, currently these new features are incompatible with C++.

Because C89 is the standard that forms the C subset of C++, and because it is the version of C that the vast majority of C programmers currently know, it is the version of C discussed in Part I. Thus, when the term *C* is used, take it to mean the C defined by C89. However, important differences between C89 and C99 that relate specifically to C++ are noted, such as when C99 adds a feature that improves compatibility with C++.

C Is a Middle-Level Language

C is often called a *middle-level* computer language. This does not mean that C is less powerful, harder to use, or less developed than a high-level language such as BASIC or Pascal; nor does it imply that C has the cumbersome nature of assembly language (and its associated troubles). Rather, C is thought of as a middle-level language because it combines the best elements of high-level languages with the control and flexibility of assembly language. Table 1-1 shows how C fits into the spectrum of computer languages.

As a middle-level language, C allows the manipulation of bits, bytes, and addresses—the basic elements with which the computer functions. Despite this fact, C code is also

Highest level	Ada
	Modula-2
	Pascal
	COBOL
	FORTRAN
	BASIC
Middle level	Java
	C#
	C++
	C
	Forth
	Macro-assembler
Lowest level	Assembler

Table 1-1. *C's Place in the World of Languages*

portable. *Portability* means that it is easy to adapt software written for one type of computer or operating system to another. For example, if you can easily convert a program written for UNIX so that it runs under Windows, that program is portable.

All high-level programming languages support the concept of data types. A *data type* defines a set of values that a variable can store along with a set of operations that can be performed on that variable. Common data types are integer, character, and real. Although C has five basic built-in data types, it is not a strongly typed language, as are Pascal and Ada. C permits almost all type conversions. For example, you may freely intermix character and integer types in an expression.

Unlike a high-level language, C performs almost no run-time error checking. For example, no check is performed to ensure that array boundaries are not overrun. These types of checks are the responsibility of the programmer.

In the same vein, C does not demand strict type compatibility between a parameter and an argument. As you may know from your other programming experience, a high-level computer language will typically require that the type of an argument be (more or less) exactly the same type as the parameter that will receive the argument. However, such is not the case for C. Instead, C allows an argument to be of any type so long as it can be reasonably converted into the type of the parameter. Further, C provides all of the automatic conversions to accomplish this.

C is special in that it allows the direct manipulation of bits, bytes, words, and pointers. This makes it well suited for system-level programming, where these operations are common.

Another important aspect of C is that it has only a few keywords, which are the commands that make up the C language. For example, C89 defines only 32 keywords, with C99 adding just another 5. Some computer languages have several times more. For comparison, most versions of BASIC have well over 100 keywords!

C Is a Structured Language

In your previous programming experience, you may have heard the term *block-structured* applied to a computer language. Although the term block-structured language does not strictly apply to C, C is commonly referred to simply as a *structured* language. It has many similarities to other structured languages, such as ALGOL, Pascal, and Modula-2.

Note *The reason that C (and C++) is not, technically, a block-structured language is that block-structured languages permit procedures or functions to be declared inside other procedures or functions. However, since C does not allow the creation of functions within functions, it cannot formally be called block-structured.*

The distinguishing feature of a structured language is *compartmentalization* of code and data. This is the ability of a language to section off and hide from the rest of the program all information and instructions necessary to perform a specific task. One way that you achieve compartmentalization is by using subroutines that employ local (temporary) variables. By using local variables, you can write subroutines so that the events that occur within them cause no side effects in other parts of the program.

This capability makes it easier for programs to share sections of code. If you develop compartmentalized functions, you only need to know what a function does, not how it does it. Remember, excessive use of global variables (variables known throughout the entire program) may allow bugs to creep into a program by allowing unwanted side effects. (Anyone who has programmed in standard BASIC is well aware of this problem.)

Note *The concept of compartmentalization is greatly expanded by C++. Specifically, in C++, one part of your program may tightly control which other parts of your program are allowed access.*

A structured language allows you a variety of programming possibilities. It directly supports several loop constructs, such as **while**, **do-while**, and **for**. In a structured language, the use of **goto** is either prohibited or discouraged and is not the common form of program control (as is the case in standard BASIC and traditional FORTRAN, for example). A structured language allows you to place statements anywhere on a line and does not require a strict field concept (as some older FORTRANs do).

Here are some examples of structured and nonstructured languages:

Nonstructured	Structured
FORTRAN	Pascal
BASIC	Ada
COBOL	Java
	C#
	C++
	C
	Modula-2

Structured languages tend to be modern. In fact, a mark of an old computer language is that it is nonstructured. Today, few programmers would consider using a nonstructured language for serious, new programs.

Note *New versions of many older languages have attempted to add structured elements. BASIC is an example—in particular Visual Basic by Microsoft. However, the shortcomings of these languages can never be fully mitigated because they were not designed with structured features from the beginning.*

C's main structural component is the function—C's stand-alone subroutine. In C, functions are the building blocks in which all program activity occurs. They allow you to define and code separately the separate tasks in a program, thus allowing your programs to be modular. After you have created a function, you can rely on it to work properly in various situations without creating side effects in other parts of the program. Being

able to create stand-alone functions is extremely critical in larger projects where one programmer's code must not accidentally affect another's code.

Another way to structure and compartmentalize code in C is through the use of code blocks. A *code block* is a logically connected group of program statements that is treated as a unit. In C, you create a code block by placing a sequence of statements between opening and closing curly braces. In this example,

```
if (x < 10)  {
  printf("Too low, try again.\n");
  scanf("%d", &x);
}
```

the two statements after the **if** and between the curly braces are both executed if **x** is less than 10. These two statements together with the braces represent a code block. They are a logical unit: one of the statements cannot execute without the other executing also. Code blocks allow many algorithms to be implemented with clarity, elegance, and efficiency. Moreover, they help the programmer better conceptualize the true nature of the algorithm being implemented.

C Is a Programmer's Language

Surprisingly, not all computer programming languages are for programmers. Consider the classic examples of nonprogrammer languages, COBOL and BASIC. COBOL was designed not to better the programmer's lot, not to improve the reliability of the code produced, and not even to improve the speed with which code can be written. Rather, COBOL was designed, in part, to enable nonprogrammers to read and presumably (however unlikely) to understand the program. BASIC was created essentially to allow nonprogrammers to program a computer to solve relatively simple problems.

In contrast, C was created, influenced, and field-tested by working programmers. The end result is that C gives the programmer what the programmer wants: few restrictions, few complaints, block structures, stand-alone functions, and a compact set of keywords. By using C, you can nearly achieve the efficiency of assembly code combined with the structure of ALGOL or Modula-2. It is no wonder that C and C++ are easily two of the most popular languages among topflight professional programmers.

The fact that you can often use C in place of assembly language is a major factor in its popularity among programmers. Assembly language uses a symbolic representation of the actual binary code that the computer executes directly. Each assembly-language operation maps into a single task for the computer to perform. Although assembly language gives programmers the potential to accomplish tasks with maximum flexibility and efficiency, it is notoriously difficult to work with when developing and debugging a program. Furthermore, since assembly language is unstructured, the final program tends to be spaghetti code—a tangled mess of jumps, calls, and indexes. This lack of structure makes assembly-language programs difficult to read, enhance, and maintain.

Perhaps more important, assembly-language routines are not portable between machines with different central processing units (CPUs).

Initially, C was used for systems programming. A *systems program* forms a portion of the operating system of the computer or its support utilities. For example, the following are usually called systems programs:

- Operating systems
- Interpreters
- Editors
- Compilers
- File utilities
- Performance enhancers
- Real-time executives
- Device drivers

As C grew in popularity, many programmers began to use it to program all tasks because of its portability and efficiency—and because they liked it! At the time of its creation, C was a much longed-for, dramatic improvement in programming languages. Of course, C++ has carried on this tradition.

With the advent of C++, some thought that C as a distinct language would die out. Such has not been the case. First, not all programs require the application of the object-oriented programming features provided by C++. For example, applications such as embedded systems are still typically programmed in C. Second, much of the world still runs on C code, and those programs will continue to be enhanced and maintained. While C's greatest legacy is as the foundation for C++, C will continue to be a vibrant, widely used language for many years to come.

The Form of a C Program

Table 1-2 lists the 32 keywords that, combined with the formal C syntax, form C89, the C subset of C++. All are, of course, also keywords in C++.

In addition, many compilers have added several keywords that better exploit their operating environment. For example, several compilers include keywords to manage the memory organization of the 8086 family of processors, to support inter-language programming, and to access interrupts. Here is a list of some commonly used extended keywords:

asm	_cs	_ds	_es
_ss	cdecl	far	huge
interrupt	near	pascal	

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 1-2. *The 32 Keywords Defined by the C Subset of C++*

Your compiler may also support other extensions that help it take better advantage of its specific environment.

Notice that all of the keywords are lowercase. C/C++ is *case-sensitive*. Thus, in a C/C++ program, uppercase and lowercase are different. This means that **else** is a keyword, while **ELSE** is not. You may not use a keyword for any other purpose in a program— that is, you may not use it as a variable or function name.

All C programs consist of one or more functions. The only function that must be present is called **main( )**, which is the first function called when program execution begins. In well-written C code, **main( )** contains what is, in essence, an outline of what the program does. The outline is composed of function calls. Although **main( )** is not a keyword, treat it as if it were. For example, don't try to use **main** as the name of a variable because you will probably confuse the compiler.

The general form of a C program is illustrated in Figure 1-1, where **f1( )** through **fN( )** represent user-defined functions.

The Library and Linking

Technically speaking, you can create a useful, functional C or C++ program that consists solely of the statements that you actually created. However, this is quite rare because neither C nor C++ provides any keywords that perform such things as I/O operations, high-level mathematical computations, or character handling. As a result, most programs include calls to various functions contained in the *standard library*.

All C++ compilers come with a standard library of functions that perform most commonly needed tasks. Standard C++ specifies a minimal set of functions that will be supported by all compilers. However, your compiler will probably contain many other functions. For example, the standard library does not define any graphics functions, but your compiler will probably include some.

```
global declarations
return-type main (parameter list)
{
 statement sequence
}
return-type f1 (parameter list)
{
 statement sequence
}
return-type f2 (parameter list)
{
 statement sequence
}
.
.
.
return-type fN(parameter list)
{
 statement sequence
}
```

Figure 1-1. *The general form of a C program*

The C++ standard library can be divided into two halves: the standard function library and the class library. The standard function library is inherited from the C language. C++ supports the entire function library defined by C89. Thus, all of the standard C functions are available for use in C++ programs that you write.

In addition to the standard function library, C++ also defines its own class library. The class library provides object-oriented routines that your programs may use. It also defines the Standard Template Library (STL), which offers off-the-shelf solutions to a variety of programming problems. Both the class library and the STL are discussed later in this book. In Part One, only the standard function library is used, since it is the only one that is also defined by C.

The standard function library contains most of the general-purpose functions that you will use. When you call a library function, the compiler "remembers" its name. Later, the linker combines the code you wrote with the object code for the library function, which is found in the standard library. This process is called *linking*. Some compilers have their own linker, while others use the standard linker supplied by your operating system.

The functions in the library are in *relocatable* format. This means that the memory addresses for the various machine-code instructions have not been absolutely defined—only offset information has been kept. When your program links with the functions in the standard library, these memory offsets are used to create the actual addresses used. Several technical manuals and books explain this process in more detail. However, you do not need any further explanation of the actual relocation process to program in C++.

Many of the functions that you will need as you write programs are in the standard library. They act as building blocks that you combine. If you write a function that you will use again and again, you can place it into a library, too.

Separate Compilation

Most short programs are completely contained within one source file. However, as a program's length grows, so does its compile time (and long compile times make for short tempers). Hence, C/C++ allows a program to be contained in multiple files and lets you compile each file separately. Once you have compiled all files, they are linked, along with any library routines, to form the complete object code. The advantage of separate compilation is that if you change the code of one file, you do not need to recompile the entire program. On all but the most simple projects, this saves a substantial amount of time. The user documentation to your C/C++ compiler will contain instructions for compiling multiple-file programs.

Understanding the .C and .CPP File Extensions

The programs in Part One of this book are, of course, valid C++ programs and can be compiled using any modern C++ compiler. They are also valid C programs and can be compiled using a C compiler. Thus, if you are called upon to write C programs, the programs shown in Part One qualify as examples. Traditionally, C programs use the file extension **.C** and C++ programs use the extension **.CPP**. A C++ compiler uses the file extension to determine what type of program it is compiling. This is important because the compiler assumes that any program using the **.C** extension is a C program and that any file using **.CPP** is a C++ program. Unless explicitly noted otherwise, you may use either extension for the programs in Part One. However, the programs in the rest of this book will require **.CPP**.

One last point: Although C is a subset of C++, there are a few minor differences between the two languages and in a few cases, you may need to compile a C program *as a C program* (using the **.C** extension). Any instances of this will be noted.