The C++ Programming Language

Bjarne Stroustrup

The C++ Programming Language

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey

ADDISON-WESLEY PUBLISHING COMPANY

Reading, Massachusetts • Menlo Park, California
Don Mills, Ontario • Wokingham, England • Amsterdam
Sydney • Singapore • Tokyo • Mexico City
Bogotá • Santiago • San Juan

This book is in the **Addison-Wesley Series in Computer Science**

Michael A. Harrison
Consulting Editor

Library of Congress Cataloging-in-Publication Data

Stroustrup, Bjarne.
The C++ programming language.

Includes bibliographies and index.
1. C++ (Computer program language) I. Title.
II. Title: C plus plus programming language.
QA76.73.C153S77 1986 005.13'3 85-20087
ISBN 0-201-12078-X

This book was typeset in Times Roman and Courier by the author, using a Mergenthaler Linotron 202 phototypesetter driven by a VAX-11/750 running the 8th Edition of the UNIX operating system.

ABCDEFGHIJK-DO-898765

Preface

Language shapes the way we think,
and determines what we can think about.
– B.L.Whorf

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types. A programmer can partition an application into manageable pieces by defining new types that closely match the concepts of the application. This technique for program construction is often called *data abstraction*. Objects of some user-defined types contain type information. Such objects can be used conveniently and safely in contexts in which their type cannot be determined at compile time. Programs using objects of such types are often called *object based*. When used well, these techniques result shorter, easier to understand, and easier to maintain programs.

The key concept in C++ is *class*. A class is a user-defined type. Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user-defined types, dynamic typing, user-controlled memory management, and mechanisms for overloading operators. C++ provides much better facilities for type checking and for expressing modularity than C does. It also contains improvements that are not directly related to classes, including symbolic constants, inline substitution of functions, default function arguments, overloaded function names, free store management operators, and a reference type. C++ retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.). This allows the user-defined types to be implemented with a pleasing degree of efficiency.

C++ and its standard libraries are designed for portability. The current implementation will run on most systems that support C. C libraries can be used from a C++ program, and most tools that support programming in C can be used with C++.

This book is primarily intended to help serious programmers learn the language and use it for nontrivial projects. It provides a complete description of C++, many complete examples, and many more program fragments.

Acknowledgments

C++ could never have matured without the constant use, suggestions, and constructive criticism of many friends and colleagues. In particular, Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwartz, and Jon Shopiro provided important ideas for development of the language. Dave Presotto wrote the current implementation of the stream I/O library.

In addition, hundreds of people contributed to the development of C++ and its compiler by sending me suggestions for improvements, descriptions of problems they had encountered, and compiler errors. I can mention only a few: Gary Bishop, Andrew Hume, Tom Karzes, Victor Milenkovic, Rob Murray, Leonie Rose, Brian Schmult, and Gary Walker.

Many people have also helped with the production of this book, in particular, Jon Bentley, Laura Eaves, Brian Kernighan, Ted Kowalski, Steve Mahaney, Jon Shopiro, and the participants in the C++ course held at Bell Labs, Columbus, Ohio, June 26-27, 1985.

Contents

Notes to the Reader

"The time has come," the Walrus said,
"to speak of many things."
– L.Carroll

This chapter consists of an overview of this book, a list of references, and some ancillary notes on C++. The notes concern the history of C++, ideas that influenced the design of C++, and thoughts about programming in C++. This chapter is not an introduction: the notes are not a prerequisite for understanding the following chapters, and some notes assume knowledge of C++.

The Structure of This Book

Chapter 1 is a quick tour of the major features of C++ intended to give the reader a feel for the language. C programmers can read the first half of the chapter very quickly; it primarily covers features common to C and C++. The second half describes C++'s facilities for defining new types; a novice may postpone a more detailed study of this until after Chapters 2, 3 and 4.

Chapters 2, 3, and 4 describe features of C++ that are not involved in defining new types: the fundamental types, expressions, and control structures for C++ programs. In other words, they describe the subset of C++ that is essentially C. They go into considerably greater detail than Chapter 1, but the complete information can be found only in the reference manual. However, these chapters provide examples, opinions, recommendations, warnings, and exercises that have no place in a manual.

Chapters 5, 6, and 7 describe C++'s facilities for defining new types, features that do not have counterparts in C. Chapter 5 presents the basic class concept, showing how objects of a user-defined type can be initialized,

accessed, and finally cleaned up. Chapter 6 explains how to define unary and binary operators for a user-defined type, how to specify conversions between user-defined types, and how to specify the way every creation, deletion, and copying of a value of a user-defined type is to be handled. Chapter 7 describes the concept of a derived class, which enables a programmer to build more complex classes from simpler ones, to provide alternative interfaces to a class, and to handle objects in an efficient and type-secure manner in contexts in which their type cannot be known at compile time.

Chapter 8 presents the `ostream` and `istream` classes provided for input and output in the *standard library*. This chapter has a dual purpose; it presents a useful facility that is also a realistic example of C++ use.

Finally, the C++ reference manual is included.

References to parts of this book are of the form §2.3.4 (Chapter 2 subsection 3.4). Chapter *r* is the reference manual; for example §r.8.5.5.

Implementation Notes

At the time of writing, all C++ implementations use versions of a single compiler front-end†. It is used on a large number of architectures, including AT&T 3B, DEC VAX, IBM 370, and Motorola 68000 running versions of the UNIX operating system. The program fragments in this book were directly taken from source files that were compiled on a 3B20 running UNIX System V release 2[15], a VAX11/750 running 8th Edition UNIX[16], and a CCI Power 6/32 running BSD4.2 UNIX[17]. The language described in this book is "pure C++", but the current compiler also implements a number of "anachronisms" (described in §r.15.3) that should ease a transition from C to C++.

Exercises

Exercises can be found at the end of chapters. The exercises are mainly of the write-a-program variety. Always write enough code for a solution to be compiled and run with at least a few test cases. The exercises vary considerably in difficulty, so they are marked with an estimate of their difficulty. The scale is exponential so that if a (*1) exercise takes you about five minutes, a (*2) might take an hour, and a (*3) might take a day. The time needed to write and test a program depends more on the reader's experience than on the exercise itself. A (*1) exercise might take a day if the reader first has to get acquainted with a new computer system to run it. On the other hand, a (*5) exercise might be done in an hour by someone who happens to have the right collection of programs handy. Any book on programming in C can be used as

† C++ is available from AT&T, Software Sales and Marketing, PO Box 25000, Greensboro, NC 27420, USA (telephone 800-828-UNIX) or from your local sales organization for the UNIX System.

a source of exercises for Chapters 2-4. Aho et. al.[1] present many common data structures and algorithms in terms of abstract data types. It can therefore be used as a source of exercises for Chapters 5-7. However, the language used in that book lacks both member functions and derived classes. Consequently, the user-defined types can often be expressed more elegantly in C++.

Design Notes

Simplicity was an important design criterion; where there was a choice between simplifying the manual and other documentation or simplifying the compiler, the former was chosen. Great importance was also attached to retaining compatibility with C; this precluded cleaning up C syntax.

C++ has no high-level data types and no high-level primitive operations. For example, there is no matrix type with an inversion operator or a string type with a concatenation operator. If a user wants such a type, it can be defined in the language itself. In fact, defining a new general-purpose or application-specific type is the most fundamental programming activity in C++. A well designed user-defined type differs from a built-in type only in the way it is defined and not in the way it is used.

Features that would incur run-time or memory overheads even when not used were avoided. For example, ideas that would make it necessary to store "housekeeping information" in every object were rejected; if a user declares a structure consisting of two 16-bit quantities, that structure will fit into a 32-bit register.

C++ was designed to be used in a rather traditional compilation and run-time environment, the C programming environment on the UNIX system. Facilities such as exception handling or concurrent programming that require nontrivial loader and run-time support are not included in C++. Consequently, a C++ implementation can be very easily ported. There are, however, good reasons for using C++ in an environment with significantly more support available. Facilities such as dynamic loading, incremental compilation, and a database of type definitions can be put to good use without affecting the language.

C++ types and data-hiding features rely on compile-time analysis of programs to prevent accidental corruption of data. They do not provide secrecy or protection against someone deliberately breaking the rules. They can, however, be used freely without incurring run-time or space overheads.

Historical Note

Clearly C++ owes most to C[7]. C is retained as a subset, and so is C's emphasis on facilities that are low-level enough to cope with the most demanding systems programming tasks. C in turn owes much to its predecessor BCPL[9]; in fact, BCPL's // comment convention has been (re)introduced in

C++. If you know BCPL you will notice that C++ still lacks a `$valof`. The other main source of inspiration was Simula67[2,3]; the class concept (with derived classes and virtual functions) was borrowed from it. The Simula67 `inspect` statement was deliberately not introduced into C++. The reason for that is to encourage modularity through the use of virtual functions. C++'s facility for overloading operators and the freedom to place a declaration wherever a statement can occur resembles Algol68[14].

The name C++ is a quite recent invention (summer of 1983). Earlier versions of the language collectively known as "C with Classes"[13] have been in use since 1980. The language was originally invented because the author wanted to write some event-driven simulations for which Simula67 would have been ideal, except for efficiency considerations. "C with Classes" was used for major simulation projects in which the facilities for writing programs that use (only) minimal time and space were severely tested. "C with Classes" lacked operator overloading, references, virtual functions, and many details. C++ was first installed outside the author's research group in July, 1983; quite a few current C++ features had not yet been invented, however.

The name C++ was coined by Rick Mascitti. The name signifies the evolutionary nature of the changes from C. "++" is the C increment operator. The slightly shorter name C+ is a syntax error; it has also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language is not called D, since it is an extension of C and does not attempt to remedy problems by removing features. For yet another interpretation of the name C++, see the appendix of Orwell[8].

C++ was primarily designed so that the author and his friends would not have to program in assembler, C, or various modern high-level languages. Its main purpose is to make writing good programs easier and more pleasant for the individual programmer. There never was a C++ paper design; design, documentation, and implementation went on simultaneously. Naturally, the C++ front-end is written in C++. There never was a "C++ project" either, or a "C++ design committee". Throughout, C++ evolved, and continues to evolve, to cope with problems encountered by users, and through discussions between the author and his friends and colleagues.

C was chosen as the base language for C++ because it (1) is versatile, terse, and relatively low-level; (2) is adequate for most system programming tasks; (3) runs everywhere and on everything; and (4) fits into the UNIX programming environment. C has its problems, but a language designed from scratch would have some too, and we know C's problems. Most important, working with C enabled "C with Classes" to be a useful (if awkward) tool within months of the first thought of adding Simula-like classes to C.

As C++ became more widely used, and as the facilities it provided over and above those of C became more significant, the question of whether to retain compatibility was raised again and again. Clearly some problems could be avoided if some of the C heritage was rejected (see, for example, Sethi[12]).

This was not done because (1) there are millions of lines of C code that might benefit from C++, provided that a complete rewrite from C to C++ were unnecessary; (2) there are hundreds of thousands of lines of library functions and utility software code written in C that could be used from/on C++ programs provided C++ were completely link compatible and syntactically very similar to C; (3) there are tens of thousands of programmers who know C and therefore need only learn to use the new features of C++ and not relearn the basics; and (4) since C++ and C will be used on the same systems by the same people for years, the differences should be either very large or very small to minimize mistakes and confusion. Lately, the definition of C++ has been revised to ensure that any construct that is both legal C and legal C++ actually has the same meaning in both languages.

The C language has itself evolved over the last few years, partly under the influence of the development of C++ (see Rosler[11]). The preliminary draft ANSI C standard[10] contains a function declaration syntax borrowed from "C with classes." Borrowing works both ways; for example, the `void*` pointer type was invented for ANSI C and first implemented in C++. When the ANSI standard has developed a bit further, it will be time to review C++ to remove gratuitous incompatibilities. For example, the preprocessor (§r.11) will be modernized, and the rules for doing floating point arithmetic will probably have to be adjusted. That should not be painful; both C and ANSI C are very close to being subsets of C++ (see §r.15).

Efficiency and Structure

C++ was developed from the C programming language and with very few exceptions retains C as a subset. The base language, the C subset of C++, is designed so that there is a very close correspondence between its types, operators, and statements and the objects computers deal with directly: numbers, characters, and addresses. Except for the free store operators `new` and `delete`, individual C++ expressions and statements typically need no hidden run-time support or subroutines.

C++ uses the same function call and return sequences as C. When even this relatively efficient mechanism is too expensive, a C++ function can be substituted inline, thus enjoying the notational convenience of functions without run-time overhead.

One of the original aims for C was to replace assembly coding for the most demanding systems programming tasks. When C++ was designed, care was taken not to compromise the gains in this area. The difference between C and C++ is primarily in the degree of emphasis on types and structure. C is expressive and permissive. C++ is even more expressive, but to gain that increase in expressiveness, the programmer must pay more attention to the types of objects. Knowing the types of objects, the compiler can deal correctly with expressions when the programmer would otherwise have had to specify

operations in painful detail. Knowing the types of objects also enables the compiler to detect errors that would otherwise have persisted until testing. Note that using the type system to get function argument checking, to protect data from accidental corruption, to provide new types, to provide new operators, etc., does not in itself increase run-time or space overheads.

The emphasis on structure in the design of C++ reflects the increase in the scale of programs written since C was designed. You can make a small program (less than 1000 lines) work through brute force even when breaking every rule of good style. For a larger program, this is simply not so. If the structure of a 10,000 line program is bad, you will find that new errors are introduced as fast as old ones are removed. C++ was designed to enable larger programs to be structured in a rational way so that it would not be unreasonable for a single person to cope with up to 25,000 lines of code. Much larger programs exist, but the ones that work generally turn out to consist of many nearly independent parts, each one well below the limits previously mentioned. Naturally, the difficulty of writing and maintaining a program depends on the complexity of the application and not simply on the number of lines of program text, so the exact numbers used to express the preceding ideas should not be taken too seriously.

However, not every piece of code can be well structured, hardware independent, easy to read, etc. C++ possesses features that are intended for manipulating hardware facilities in a direct and efficient way without regard for safety or ease of comprehension. It also possesses facilities for hiding such code behind elegant and safe interfaces.

This book emphasizes techniques for providing general-purpose facilities, generally useful types, libraries, etc. These techniques will serve programmers of small programs as well as programmers of large ones. Furthermore, since all nontrivial programs consist of many semi-independent parts, the techniques for writing such parts serve programmers of both systems and applications.

One might suspect that specifying a program using a more detailed type structure would lead to a larger program source text. With C++ this is not so; a C++ program declaring functions argument types, using classes, etc., is typically a bit shorter than the equivalent C program not using these facilities.

Philosophical Note

A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed and a set of concepts for the programmer to use when thinking about what can be done. The first aspect ideally requires a language that is "close to the machine", so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second aspect ideally requires a language that is "close to the problem to be solved" so that the concepts of a solution can be expressed

directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.

The connection between the language in which we think/program and the problems and solutions we can imagine is very close. For this reason restricting language features with the intent of eliminating programmer errors is at best dangerous. As with natural languages, there are great benefits from being at least bilingual. The language provides a programmer with a set of conceptual tools; if these are inadequate for a task, they will simply be ignored. For example, seriously restricting the concept of a pointer simply forces the programmer to use a vector plus integer arithmetic to implement structures, pointers, etc. Good design and the absence of errors cannot be guaranteed by mere language features.

The type system should be especially helpful for nontrivial tasks. The C++ class concept has, in fact, proven itself as a powerful conceptual tool.

Thinking about Programming in C++

Ideally one approaches the task of designing a program in three stages: first gain a clear understanding of the problem, then identify the key concepts involved in a solution, and finally express that solution in a program. However, the details of the problem and the concepts of the solution often become clearly understood only through the effort to express them in the program – this is where the choice of programming language matters.

In most applications there are concepts that are not easily represented in a program as either one of the fundamental types or as a function without associated static data. Given such a concept, declare a class to represent it in the program. A class is a type; that is, it specifies how objects of its class behave: how they are created, how they can be manipulated, how they are destroyed. A class also specifies how objects are represented, but at the early stages of the design of a program, that is not (should not be) the major concern. The key to writing a good program is to design classes so that each cleanly represents a single concept. Often this means that the programmer must focus on the questions: How are objects of this class created? Can objects of this class be copied and/or destroyed? What operations can be done on such objects? If there are no good answers to such questions, the concept probably wasn't "clean" in the first place, and it might be a good idea to think a bit more about the problem and the proposed solution instead of immediately starting to "code around" the problems.

The concepts that are easiest to deal with are the ones that have a traditional mathematical formalism: numbers of all sorts, sets, geometric shapes, etc. There really ought to be standard libraries of classes representing such concepts, but this is not the case at the time of writing. C++ is still young, and its libraries have not yet matured to the same degree as the language itself.

A concept does not exist in a vacuum; there are always clusters of related

concepts. Organizing the relationship between classes in a program, that is, determining the exact relationship between the different concepts involved in a solution, is often harder than laying out the individual classes in the first place. The result had better not be a muddle in which every class (concept) depends on every other. Consider two classes, A and B: Relationships such as "A calls functions from B," "A creates Bs," and "A has a B member" seldom cause major problems, and relationships such as "A uses data from B" can typically be eliminated (simply don't use public data members). The trouble spots are most often relations that are naturally expressed as "A is a B and ... ."

One of the most powerful intellectual tools for managing complexity is hierarchical ordering; that is, organizing related concepts into a tree structure with the most general concept at the root. In C++, derived classes represent such structures. A program can often be organized as a set of trees (a forest?). That is, the programmer specifies a number of base classes, each with its own set of derived classes. Virtual functions (§7.2.8) can often be used to define a set of operations for the most general version of a concept (a base class). When necessary, the interpretation of these operations can be refined for particular special cases (derived classes).

Naturally, this organization has its limits. In particular, a set of concepts is sometimes better organized as a directed acyclic graph in which a concept can directly depend on more than one other concept; for example, "A is a B and a C and ... ." There is no direct support for this in C++, but such relations can be represented with some loss of elegance and a bit of extra work (§7.2.5).

Sometimes even a directed acyclic graph seems insufficient for organizing the concepts of a program; some concepts seem to be inherently mutually dependent. If a set of mutually dependent classes is so small that it is easy to understand, the cyclic dependencies need not be a problem. The idea of `friend` classes (§5.4.1) can be used to represent sets of mutually dependent classes in C++.

If you can organize the concepts of a program only into a general graph (and not a tree or a directed acyclic graph), and if you cannot localize the mutual dependencies, then you are most likely in a predicament that no programming language can help you out of. Unless you can conceive of some easily stated relationships between the basic concepts, the program is likely to become unmanageable.

Remember that much programming can be simply and clearly done using only primitive types, data structures, plain functions, and a few classes from a standard library. The whole apparatus involved in defining new types should not be used except when there is a real need.

The question "How does one write good programs in C++?" is very similar to the question "How does one write good English prose?" There are two kinds of answers: "Know what you want to say" and "Practice. Imitate good writing." Both kinds of advice appear to be as appropriate for C++ as they are for English – and as hard to follow.