1

Notes to the Reader

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

Structure of this book — how to learn C++ — the design of C++ — efficiency and structure — philosophical note — historical note — what C++ is used for — C and C++ — suggestions for C programmers — suggestions for C++ programmers — thoughts about programming in C++ — advice — references.

1.1 The Structure of This Book

This book consists of six parts:

Introduction: Chapters 1 through 3 give an overview of the C++ language, the key programming styles it supports, and the C++ standard library.

Part I: Chapters 4 through 9 provide a tutorial introduction to C++'s built-in types and the basic facilities for constructing programs out of them.

Part II: Chapters 10 through 15 are a tutorial introduction to object-oriented and generic programming using C++.

Part III: Chapters 16 through 22 present the C++ standard library.

Part IV: Chapters 23 through 25 discuss design and software development issues.

Appendices: Appendices A through E provide language-technical details.

Chapter 1 provides an overview of this book, some hints about how to use it, and some background information about C++ and its use. You are encouraged to skim through it, read what appears interesting, and return to it after reading other parts of the book.

Chapters 2 and 3 provide an overview of the major concepts and features of the C++ programming language and its standard library. Their purpose is to motivate you to spend time on fundamental concepts and basic language features by showing what can be expressed using the complete

C++ language. If nothing else, these chapters should convince you that C++ isn't (just) C and that C++ has come a long way since the first and second editions of this book. Chapter 2 gives a high-level acquaintance with C++. The discussion focuses on the language features supporting data abstraction, object-oriented programming, and generic programming. Chapter 3 introduces the basic principles and major facilities of the standard library. This allows me to use standard library facilities in the following chapters. It also allows you to use library facilities in exercises rather than relying directly on lower-level, built-in features.

The introductory chapters provide an example of a general technique that is applied throughout this book: to enable a more direct and realistic discussion of some technique or feature, I occasionally present a concept briefly at first and then discuss it in depth later. This approach allows me to present concrete examples before a more general treatment of a topic. Thus, the organization of this book reflects the observation that we usually learn best by progressing from the concrete to the abstract – even where the abstract seems simple and obvious in retrospect.

Part I describes the subset of C++ that supports the styles of programming traditionally done in C or Pascal. It covers fundamental types, expressions, and control structures for C++ programs. Modularity – as supported by namespaces, source files, and exception handling – is also discussed. I assume that you are familiar with the fundamental programming concepts used in Part I. For example, I explain C++'s facilities for expressing recursion and iteration, but I do not spend much time explaining how these concepts are useful.

Part II describes C++'s facilities for defining and using new types. Concrete and abstract classes (interfaces) are presented here (Chapter 10, Chapter 12), together with operator overloading (Chapter 11), polymorphism, and the use of class hierarchies (Chapter 12, Chapter 15). Chapter 13 presents templates, that is, C++'s facilities for defining families of types and functions. It demonstrates the basic techniques used to provide containers, such as lists, and to support generic programming. Chapter 14 presents exception handling, discusses techniques for error handling, and presents strategies for fault tolerance. I assume that you either aren't well acquainted with object-oriented programming and generic programming or could benefit from an explanation of how the main abstraction techniques are supported by C++. Thus, I don't just present the language features supporting the abstraction techniques; I also explain the techniques themselves. Part IV goes further in this direction.

Part III presents the C++ standard library. The aim is to provide an understanding of how to use the library, to demonstrate general design and programming techniques, and to show how to extend the library. The library provides containers (such as ***list***, ***vector***, and ***map***; Chapter 16, Chapter 17), standard algorithms (such as ***sort***, ***find***, and ***merge***; Chapter 18, Chapter 19), strings (Chapter 20), Input/Output (Chapter 21), and support for numerical computation (Chapter 22).

Part IV discusses issues that arise when C++ is used in the design and implementation of large software systems. Chapter 23 concentrates on design and management issues. Chapter 24 discusses the relation between the C++ programming language and design issues. Chapter 25 presents some ways of using classes in design.

Appendix A is C++'s grammar, with a few annotations. Appendix B discusses the relation between C and C++ and between Standard C++ (also called ISO C++ and ANSI C++) and the versions of C++ that preceded it. Appendix C presents some language-technical examples. Appendix D explains the standard library's facilities supporting internationalization. Appendix E discusses the exception-safety guarantees and requirements of the standard library.

1.1.1 Examples and References

This book emphasizes program organization rather than the writing of algorithms. Consequently, I avoid clever or harder-to-understand algorithms. A trivial algorithm is typically better suited to illustrate an aspect of the language definition or a point about program structure. For example, I use a Shell sort where, in real code, a quicksort would be better. Often, reimplementation with a more suitable algorithm is an exercise. In real code, a call of a library function is typically more appropriate than the code used here for illustration of language features.

Textbook examples necessarily give a warped view of software development. By clarifying and simplifying the examples, the complexities that arise from scale disappear. I see no substitute for writing realistically-sized programs for getting an impression of what programming and a programming language are really like. This book concentrates on the language features, the basic techniques from which every program is composed, and the rules for composition.

The selection of examples reflects my background in compilers, foundation libraries, and simulations. Examples are simplified versions of what is found in real code. The simplification is necessary to keep programming language and design points from getting lost in details. There are no ''cute'' examples without counterparts in real code. Wherever possible, I relegated to Appendix C language-technical examples of the sort that use variables named ***x*** and ***y***, types called ***A*** and ***B***, and functions called *f*( ) and *g*( ).

In code examples, a proportional-width font is used for identifiers. For example:

```
#include<iostream>

int main()
{
    std::cout << "Hello, new world!\n";
}
```

At first glance, this presentation style will seem ''unnatural'' to programmers accustomed to seeing code in constant-width fonts. However, proportional-width fonts are generally regarded as better than constant-width fonts for presentation of text. Using a proportional-width font also allows me to present code with fewer illogical line breaks. Furthermore, my experiments show that most people find the new style more readable after a short while.

Where possible, the C++ language and library features are presented in the context of their use rather than in the dry manner of a manual. The language features presented and the detail in which they are described reflect my view of what is needed for effective use of C++. A companion, *The Annotated C++ Language Standard*, authored by Andrew Koenig and myself, is the complete definition of the language together with comments aimed at making it more accessible. Logically, there ought to be another companion, *The Annotated C++ Standard Library*. However, since both time and my capacity for writing are limited, I cannot promise to produce that.

References to parts of this book are of the form §2.3.4 (Chapter 2, section 3, subsection 4), §B.5.6 (Appendix B, subsection 5.6), and §6.6[10] (Chapter 6, exercise 10). Italics are used sparingly for emphasis (e.g., ''a string literal is *not* acceptable''), for first occurrences of important concepts (e.g., *polymorphism*), for nonterminals of the C++ grammar (e.g., *for-statement*), and for comments in code examples. Semi-bold italics are used to refer to identifiers, keywords, and numeric values from code examples (e.g., ***counter***, ***class***, and ***1712***).

1.1.2 Exercises

Exercises are found at the ends 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 ten 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 your experience than on the exercise itself. A (∗1) exercise might take a day if you first have to get acquainted with a new computer system in order 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 a source of extra exercises for Part I. Any book on data structures and algorithms can be used as a source of exercises for Parts II and III.

1.1.3 Implementation Note

The language used in this book is ''pure C++'' as defined in the C++ standard [C++,1998]. Therefore, the examples ought to run on every C++ implementation. The major program fragments in this book were tried using several C++ implementations. Examples using features only recently adopted into C++ didn't compile on every implementation. However, I see no point in mentioning which implementations failed to compile which examples. Such information would soon be out of date because implementers are working hard to ensure that their implementations correctly accept every C++ feature. See Appendix B for suggestions on how to cope with older C++ compilers and with code written for C compilers.

1.2 Learning C++

The most important thing to do when learning C++ is to focus on concepts and not get lost in language-technical details. The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones. For this, an appreciation of programming and design techniques is far more important than an understanding of details; that understanding comes with time and practice.

C++ supports a variety of programming styles. All are based on strong static type checking, and most aim at achieving a high level of abstraction and a direct representation of the programmer's ideas. Each style can achieve its aims effectively while maintaining run-time and space efficiency. A programmer coming from a different language (say C, Fortran, Smalltalk, Lisp, ML, Ada, Eiffel, Pascal, or Modula-2) should realize that to gain the benefits of C++, they must spend time learning and internalizing programming styles and techniques suitable to C++. The same applies to programmers used to an earlier and less expressive version of C++.

Thoughtlessly applying techniques effective in one language to another typically leads to awkward, poorly performing, and hard-to-maintain code. Such code is also most frustrating to write because every line of code and every compiler error message reminds the programmer that the language used differs from ''the old language.'' You can write in the style of Fortran, C, Smalltalk, etc., in any language, but doing so is neither pleasant nor economical in a language with a different philosophy. Every language can be a fertile source of ideas of how to write C++ programs.

However, ideas must be transformed into something that fits with the general structure and type system of C++ in order to be effective in the different context. Over the basic type system of a language, only Pyrrhic victories are possible.

C++ supports a gradual approach to learning. How you approach learning a new programming language depends on what you already know and what you aim to learn. There is no one approach that suits everyone. My assumption is that you are learning C++ to become a better programmer and designer. That is, I assume that your purpose in learning C++ is not simply to learn a new syntax for doing things the way you used to, but to learn new and better ways of building systems. This has to be done gradually because acquiring any significant new skill takes time and requires practice. Consider how long it would take to learn a new natural language well or to learn to play a new musical instrument well. Becoming a better system designer is easier and faster, but not as much easier and faster as most people would like it to be.

It follows that you will be using C++ – often for building real systems – before understanding every language feature and technique. By supporting several programming paradigms (Chapter 2), C++ supports productive programming at several levels of expertise. Each new style of programming adds another tool to your toolbox, but each is effective on its own and each adds to your effectiveness as a programmer. C++ is organized so that you can learn its concepts in a roughly linear order and gain practical benefits along the way. This is important because it allows you to gain benefits roughly in proportion to the effort expended.

In the continuing debate on whether one needs to learn C before C++, I am firmly convinced that it is best to go directly to C++. C++ is safer, more expressive, and reduces the need to focus on low-level techniques. It is easier for you to learn the trickier parts of C that are needed to compensate for its lack of higher-level facilities after you have been exposed to the common subset of C and C++ and to some of the higher-level techniques supported directly in C++. Appendix B is a guide for programmers going from C++ to C, say, to deal with legacy code.

Several independently developed and distributed implementations of C++ exist. A wealth of tools, libraries, and software development environments are also available. A mass of textbooks, manuals, journals, newsletters, electronic bulletin boards, mailing lists, conferences, and courses are available to inform you about the latest developments in C++, its use, tools, libraries, implementations, etc. If you plan to use C++ seriously, I strongly suggest that you gain access to such sources. Each has its own emphasis and bias, so use at least two. For example, see [Barton,1994], [Booch,1994], [Henricson,1997], [Koenig,1997], [Martin,1995].

1.3 The Design of C++

Simplicity was an important design criterion: where there was a choice between simplifying the language definition and simplifying the compiler, the former was chosen. However, great importance was attached to retaining a high degree of compatibility with C [Koenig,1989] [Stroustrup,1994] (Appendix B); this precluded cleaning up the C syntax.

C++ has no built-in high-level data types and no high-level primitive operations. For example, the C++ language does not provide a 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, not in the way it is used. The C++ standard library described in Part III provides many examples of such types and their uses. From a user's point of view, there is little difference between a built-in type and a type provided by the standard library.

Features that would incur run-time or memory overheads even when not used were avoided in the design of C++. For example, constructs that would make it necessary to store ''housekeeping information'' in every object were rejected, so 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 traditional compilation and run-time environment, that is, the C programming environment on the UNIX system. Fortunately, C++ was never restricted to UNIX; it simply used UNIX and C as a model for the relationships between language, libraries, compilers, linkers, execution environments, etc. That minimal model helped C++ to be successful on essentially every computing platform. There are, however, good reasons for using C++ in environments that provide significantly more support. Facilities such as dynamic loading, incremental compilation, and a database of type definitions can be put to good use without affecting the language.

C++ type-checking 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 who is deliberately breaking the rules. They can, however, be used freely without incurring run-time or space overheads. The idea is that to be useful, a language feature must not only be elegant; it must also be affordable in the context of a real program.

For a systematic and detailed description of the design of C++, see [Stroustrup,1994].

1.3.1 Efficiency and Structure

C++ was developed from the C programming language and, with few exceptions, retains C as a subset. The base language, the C subset of C++, is designed to ensure a very close correspondence between its types, operators, and statements and the objects that computers deal with directly: numbers, characters, and addresses. Except for the ***new***, ***delete***, ***typeid***, ***dynamic_cast***, and ***throw*** operators and the *try-block*, individual C++ expressions and statements need no run-time support.

C++ can use the same function call and return sequences as C – or more efficient ones. When even such relatively efficient mechanisms are too expensive, a C++ function can be substituted inline, so that we can enjoy 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. However, to gain that increase in expressiveness, you must pay more attention to the types of objects. Knowing the types of objects, the compiler can deal correctly with expressions when you 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 persist until testing – or even later. Note that using the type system to check function arguments, to protect data from accidental corruption, to provide new types, to provide new operators, etc., does not increase run-time or space overheads in C++.

The emphasis on structure in C++ reflects the increase in the scale of programs written since C was designed. You can make a small program (say, 1,000 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 100,000-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 be reasonable for a single person to cope with far larger amounts of code. In addition, the aim was to have an average line of C++ code express much more than the average line of C or Pascal code. C++ has by now been shown to over-fulfill these goals.

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.

Naturally, the use of C++ for larger programs leads to the use of C++ by groups of programmers. C++'s emphasis on modularity, strongly typed interfaces, and flexibility pays off here. C++ has as good a balance of facilities for writing large programs as any language has. However, as programs get larger, the problems associated with their development and maintenance shift from being language problems to more global problems of tools and management. Part IV explores some of these issues.

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, because all nontrivial programs consist of many semi-independent parts, the techniques for writing such parts serve programmers of all applications.

You might suspect that specifying a program by using a more detailed type structure would lead to a larger program source text. With C++, this is not so. A C++ program declaring function argument types, using classes, etc., is typically a bit shorter than the equivalent C program not using these facilities. Where libraries are used, a C++ program will appear much shorter than its C equivalent, assuming, of course, that a functioning C equivalent could have been built.

1.3.2 Philosophical Note

A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides a set of concepts for the programmer to use when thinking about what can be done. The first purpose 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 purpose 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. A language provides a programmer with a set of conceptual tools; if these are inadequate for a task, they will simply be ignored. Good design and the absence of errors cannot be guaranteed merely by the presence or the absence of specific language features.

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

1.4 Historical Note

I invented C++, wrote its early definitions, and produced its first implementation. I chose and formulated the design criteria for C++, designed all its major facilities, and was responsible for the processing of extension proposals in the C++ standards committee.

Clearly, C++ owes much to C [Kernighan,1978]. Except for closing a few serious loopholes in the type system (see Appendix B), C is retained as a subset. I also retained 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 [Richards,1980]; in fact, BCPL's // comment convention was (re)introduced in C++. The other main source of inspiration for C++ was Simula67 [Dahl,1970] [Dahl,1972]; the class concept (with derived classes and virtual functions) was borrowed from it. C++'s facility for overloading operators and the freedom to place a declaration wherever a statement can occur resembles Algol68 [Woodward,1974].

Since the original edition of this book, the language has been extensively reviewed and refined. The major areas for revision were overload resolution, linking, and memory management facilities. In addition, several minor changes were made to increase C compatibility. Several generalizations and a few major extensions were added: these included multiple inheritance, ***static*** member functions, ***const*** member functions, ***protected*** members, templates, exception handling, run-time type identification, and namespaces. The overall theme of these extensions and revisions was to make C++ a better language for writing and using libraries. The evolution of C++ is described in [Stroustrup,1994].

The template facility was primarily designed to support statically typed containers (such as lists, vectors, and maps) and to support elegant and efficient use of such containers (generic programming). A key aim was to reduce the use of macros and casts (explicit type conversion). Templates were partly inspired by Ada's generics (both their strengths and their weaknesses) and partly by Clu's parameterized modules. Similarly, the C++ exception-handling mechanism was inspired partly by Ada [Ichbiah,1979], Clu [Liskov,1979], and ML [Wikström,1987]. Other developments in the 1985 to 1995 time span – such as multiple inheritance, pure virtual functions, and namespaces – were primarily generalizations driven by experience with the use of C++ rather than ideas imported from other languages.

Earlier versions of the language, collectively known as ''C with Classes'' [Stroustrup,1994], have been in use since 1980. The language was originally invented because I 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 projects in which the facilities for writing programs that use minimal time and space were severely tested. It lacked operator overloading, references, virtual functions, templates, exceptions, and many details. The first use of C++ outside a research organization started in July 1983.

The name C++ (pronounced ''see plus plus'') was coined by Rick Mascitti in the summer of 1983. 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, because it is an extension of C, and it does not attempt to remedy problems by removing features. For yet another interpretation of the name C++, see the appendix of [Orwell,1949].

C++ was designed primarily so that my friends and I would not have to program in assembler,

C, or various modern high-level languages. Its main purpose was to make writing good programs easier and more pleasant for the individual programmer. In the early years, there was no C++ paper design; design, documentation, and implementation went on simultaneously. There was no ''C++ project'' either, or a ''C++ design committee.'' Throughout, C++ evolved to cope with problems encountered by users and as a result of discussions between my friends, my colleagues, and me.

Later, the explosive growth of C++ use caused some changes. Sometime during 1987, it became clear that formal standardization of C++ was inevitable and that we needed to start preparing the ground for a standardization effort [Stroustrup,1994]. The result was a conscious effort to maintain contact between implementers of C++ compilers and major users through paper and electronic mail and through face-to-face meetings at C++ conferences and elsewhere.

AT&T Bell Laboratories made a major contribution to this by allowing me to share drafts of revised versions of the C++ reference manual with implementers and users. Because many of these people work for companies that could be seen as competing with AT&T, the significance of this contribution should not be underestimated. A less enlightened company could have caused major problems of language fragmentation simply by doing nothing. As it happened, about a hundred individuals from dozens of organizations read and commented on what became the generally accepted reference manual and the base document for the ANSI C++ standardization effort. Their names can be found in *The Annotated C++ Reference Manual* [Ellis,1989]. Finally, the X3J16 committee of ANSI was convened in December 1989 at the initiative of Hewlett-Packard. In June 1991, this ANSI (American national) standardization of C++ became part of an ISO (international) standardization effort for C++. From 1990, these joint C++ standards committees have been the main forum for the evolution of C++ and the refinement of its definition. I served on these committees throughout. In particular, as the chairman of the working group for extensions, I was directly responsible for the handling of proposals for major changes to C++ and the addition of new language features. An initial draft standard for public review was produced in April 1995. The ISO C++ standard (ISO/IEC 14882) was ratified in 1998.

C++ evolved hand-in-hand with some of the key classes presented in this book. For example, I designed complex, vector, and stack classes together with the operator overloading mechanisms. String and list classes were developed by Jonathan Shopiro and me as part of the same effort. Jonathan's string and list classes were the first to see extensive use as part of a library. The string class from the standard C++ library has its roots in these early efforts. The task library described in [Stroustrup,1987] and in §12.7[11] was part of the first ''C with Classes'' program ever written. I wrote it and its associated classes to support Simula-style simulations. The task library has been revised and reimplemented, notably by Jonathan Shopiro, and is still in extensive use. The stream library as described in the first edition of this book was designed and implemented by me. Jerry Schwarz transformed it into the iostreams library (Chapter 21) using Andrew Koenig's manipulator technique (§21.4.6) and other ideas. The iostreams library was further refined during standardization, when the bulk of the work was done by Jerry Schwarz, Nathan Myers, and Norihiro Kumagai. The development of the template facility was influenced by the ***vector***, ***map***, ***list***, and ***sort*** templates devised by Andrew Koenig, Alex Stepanov, me, and others. In turn, Alex Stepanov's work on generic programming using templates led to the containers and algorithms parts of the standard C++ library (§16.3, Chapter 17, Chapter 18, §19.2). The ***valarray*** library for numerical computation (Chapter 22) is primarily the work of Kent Budge.

1.5 Use of C++

C++ is used by hundreds of thousands of programmers in essentially every application domain. This use is supported by about a dozen independent implementations, hundreds of libraries, hundreds of textbooks, several technical journals, many conferences, and innumerable consultants. Training and education at a variety of levels are widely available.

Early applications tended to have a strong systems programming flavor. For example, several major operating systems have been written in C++ [Campbell,1987] [Rozier,1988] [Hamilton,1993] [Berg,1995] [Parrington,1995] and many more have key parts done in C++. I considered uncompromising low-level efficiency essential for C++. This allows us to use C++ to write device drivers and other software that rely on direct manipulation of hardware under real-time constraints. In such code, predictability of performance is at least as important as raw speed. Often, so is compactness of the resulting system. C++ was designed so that every language feature is usable in code under severe time and space constraints [Stroustrup,1994,§4.5].

Most applications have sections of code that are critical for acceptable performance. However, the largest amount of code is not in such sections. For most code, maintainability, ease of extension, and ease of testing is key. C++'s support for these concerns has led to its widespread use where reliability is a must and in areas where requirements change significantly over time. Examples are banking, trading, insurance, telecommunications, and military applications. For years, the central control of the U.S. long-distance telephone system has relied on C++ and every 800 call (that is, a call paid for by the called party) has been routed by a C++ program [Kamath,1993]. Many such applications are large and long-lived. As a result, stability, compatibility, and scalability have been constant concerns in the development of C++. Million-line C++ programs are not uncommon.

Like C, C++ wasn't specifically designed with numerical computation in mind. However, much numerical, scientific, and engineering computation is done in C++. A major reason for this is that traditional numerical work must often be combined with graphics and with computations relying on data structures that don't fit into the traditional Fortran mold [Budge,1992] [Barton,1994]. Graphics and user interfaces are areas in which C++ is heavily used. Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are C++ programs. In addition, some of the most popular libraries supporting X for UNIX are written in C++. Thus, C++ is a common choice for the vast number of applications in which the user interface is a major part.

All of this points to what may be C++'s greatest strength: its ability to be used effectively for applications that require work in a variety of application areas. It is quite common to find an application that involves local and wide-area networking, numerics, graphics, user interaction, and database access. Traditionally, such application areas have been considered distinct, and they have most often been served by distinct technical communities using a variety of programming languages. However, C++ has been widely used in all of those areas. Furthermore, it is able to coexist with code fragments and programs written in other languages.

C++ is widely used for teaching and research. This has surprised some who – correctly – point out that C++ isn't the smallest or cleanest language ever designed. It is, however

– clean enough for successful teaching of basic concepts,
– realistic, efficient, and flexible enough for demanding projects,

– available enough for organizations and collaborations relying on diverse development and execution environments,
– comprehensive enough to be a vehicle for teaching advanced concepts and techniques, and
– commercial enough to be a vehicle for putting what is learned into non-academic use.

C++ is a language that you can grow with.

1.6 C and C++

C was chosen as the base language for C++ because it

[1] is versatile, terse, and relatively low-level;
[2] is adequate for most systems 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. Importantly, 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, e.g., [Sethi,1981]). 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 millions of lines of library functions and utility software code written in C that could be used from/on C++ programs provided C++ were link-compatible with and syntactically very similar to C;
[3] there are hundreds 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] C++ and C will be used on the same systems by the same people for years, so the differences should be either very large or very small so as to minimize mistakes and confusion.

The definition of C++ has been revised to ensure that a construct that is both legal C and legal C++ has the same meaning in both languages (with a few minor exceptions; see §B.2).

The C language has itself evolved, partly under the influence of the development of C++ [Rosler,1984]. The ANSI C standard [C,1990] 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++. As promised in the first edition of this book, the definition of C++ has been reviewed to remove gratuitous incompatibilities; C++ is now more compatible with C than it was originally. The ideal was for C++ to be as close to ANSI C as possible – but no closer [Koenig,1989]. One hundred percent compatibility was never a goal because that would compromise type safety and the smooth integration of user-defined and built-in types.

Knowing C is not a prerequisite for learning C++. Programming in C encourages many techniques and tricks that are rendered unnecessary by C++ language features. For example, explicit type conversion (casting) is less frequently needed in C++ than it is in C (§1.6.1). However, *good* C programs tend to be C++ programs. For example, every program in Kernighan and Ritchie, *The*

C Programming Language (2nd Edition) [Kernighan,1988], is a C++ program. Experience with any statically typed language will be a help when learning C++.

1.6.1 Suggestions for C Programmers

The better one knows C, the harder it seems to be to avoid writing C++ in C style, thereby losing some of the potential benefits of C++. Please take a look at Appendix B, which describes the differences between C and C++. Here are a few pointers to the areas in which C++ has better ways of doing something than C has:

[1] Macros are almost never necessary in C++. Use ***const*** (§5.4) or ***enum*** (§4.8) to define manifest constants, ***inline*** (§7.1.1) to avoid function-calling overhead, ***template***s (Chapter 13) to specify families of functions and types, and ***namespace***s (§8.2) to avoid name clashes.

[2] Don't declare a variable before you need it so that you can initialize it immediately. A declaration can occur anywhere a statement can (§6.3.1), in *for-statement* initializers (§6.3.3), and in conditions (§6.3.2.1).

[3] Don't use ***malloc***( ). The ***new*** operator (§6.2.6) does the same job better, and instead of ***realloc***( ), try a ***vector*** (§3.8).

[4] Try to avoid ***void****, pointer arithmetic, unions, and casts, except deep within the implementation of some function or class. In most cases, a cast is an indication of a design error. If you must use an explicit type conversion, try using one of the ''new casts'' (§6.2.7) for a more precise statement of what you are trying to do.

[5] Minimize the use of arrays and C-style strings. The C++ standard library ***string*** (§3.5) and ***vector*** (§3.7.1) classes can often be used to simplify programming compared to traditional C style. In general, try not to build yourself what has already been provided by the standard library.

To obey C linkage conventions, a C++ function must be declared to have C linkage (§9.2.4).

Most important, try thinking of a program as a set of interacting concepts represented as classes and objects, instead of as a bunch of data structures with functions twiddling their bits.

1.6.2 Suggestions for C++ Programmers

By now, many people have been using C++ for a decade. Many more are using C++ in a single environment and have learned to live with the restrictions imposed by early compilers and first-generation libraries. Often, what an experienced C++ programmer has failed to notice over the years is not the introduction of new features as such, but rather the changes in relationships between features that make fundamental new programming techniques feasible. In other words, what you didn't think of when first learning C++ or found impractical just might be a superior approach today. You find out only by re-examining the basics.

Read through the chapters in order. If you already know the contents of a chapter, you can be through in minutes. If you don't already know the contents, you'll have learned something unexpected. I learned a fair bit writing this book, and I suspect that hardly any C++ programmer knows every feature and technique presented. Furthermore, to use the language well, you need a perspective that brings order to the set of features and techniques. Through its organization and examples, this book offers such a perspective.

1.7 Thinking about Programming in C++

Ideally, you approach the task of designing a program in three stages. First, you gain a clear understanding of the problem (analysis), then you identify the key concepts involved in a solution (design), and finally you express that solution in a program (programming). However, the details of the problem and the concepts of the solution often become clearly understood only through the effort to express them in a program and trying to get it to run acceptably. This is where the choice of programming language matters.

In most applications, there are concepts that are not easily represented as one of the fundamental types or as a function without associated data. Given such a concept, declare a class to represent it in the program. A C++ class is a type. That is, it specifies how objects of its class behave: how they are created, how they can be manipulated, and how they are destroyed. A class may also specify how objects are represented, although in the early stages of the design of a program that should not be the major concern. The key to writing good programs is to design classes so that each cleanly represents a single concept. Often, this means that you must focus on questions such as: How are objects of this class created? Can objects of this class be copied and/or destroyed? What operations can be applied to such objects? If there are no good answers to such questions, the concept probably wasn't ''clean'' in the first place. It might then be a good idea to think more about the problem and its 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. Text-oriented I/O, strings, basic containers, the fundamental algorithms on such containers, and some mathematical classes are part of the standard C++ library (Chapter 3, §16.1.2). In addition, a bewildering variety of libraries supporting general and domain-specific concepts are available.

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, while relationships such as ''A uses data from B'' can typically be eliminated.

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 as the root. In C++, derived classes represent such structures. A program can often be organized as a set of trees or directed acyclic graphs of classes. That is, the programmer specifies a number of base classes, each with its own set of derived classes. Virtual functions (§2.5.5, §12.2.6) can often be used to define 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).

Sometimes even a directed acyclic graph seems insufficient for organizing the concepts of a program; some concepts seem to be inherently mutually dependent. In that case, we try to localize cyclic dependencies so that they do not affect the overall structure of the program. If you cannot eliminate or localize such 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.

One of the best tools for untangling dependency graphs is the clean separation of interface and implementation. Abstract classes (§2.5.4, §12.3) are C++'s primary tool for doing that.

Another form of commonality can be expressed through templates (§2.7, Chapter 13). A class template specifies a family of classes. For example, a list template specifies ''list of T,'' where ''T'' can be any type. Thus, a template is a mechanism for specifying how one type is generated given another type as an argument. The most common templates are container classes such as lists, vectors, and associative arrays (maps) and the fundamental algorithms using such containers. It is usually a mistake to express parameterization of a class and its associated functions with a type using inheritance. It is best done using templates.

Remember that much programming can be simply and clearly done using only primitive types, data structures, plain functions, and a few library classes. 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 answers: ''Know what you want to say'' and ''Practice. Imitate good writing.'' Both appear to be as appropriate for C++ as they are for English – and as hard to follow.

1.8 Advice

Here is a set of ''rules'' you might consider while learning C++. As you get more proficient you can evolve them into something suitable for your kind of applications and your style of programming. They are deliberately very simple, so they lack detail. Don't take them too literally. To write a good program takes intelligence, taste, and patience. You are not going to get it right the first time. Experiment!

[1] When you program, you create a concrete representation of the ideas in your solution to some problem. Let the structure of the program reflect those ideas as directly as possible:
  [a] If you can think of ''it'' as a separate idea, make it a class.
  [b] If you can think of ''it'' as a separate entity, make it an object of some class.
  [c] If two classes have a common interface, make that interface an abstract class.
  [d] If the implementations of two classes have something significant in common, make that commonality a base class.
  [e] If a class is a container of objects, make it a template.
  [f] If a function implements an algorithm for a container, make it a template function implementing the algorithm for a family of containers.
  [g] If a set of classes, templates, etc., are logically related, place them in a common namespace.

[2] When you define either a class that does not implement either a mathematical entity like a matrix or a complex number or a low-level type such as a linked list:
  [a] Don't use global data (use members).
  [b] Don't use global functions.
  [c] Don't use public data members.
  [d] Don't use friends, except to avoid [a] or [c].
  [e] Don't put a ''type field'' in a class; use virtual functions.
  [f] Don't use inline functions, except as a significant optimization.

More specific or detailed rules of thumb can be found in the ''Advice'' section of each chapter. Remember, this advice is only rough rules of thumb, not immutable laws. A piece of advice should be applied only ''where reasonable.'' There is no substitute for intelligence, experience, common sense, and good taste.

I find rules of the form ''never do this'' unhelpful. Consequently, most advice is phrased as suggestions of what to do, while negative suggestions tend not to be phrased as absolute prohibitions. I know of no major feature of C++ that I have not seen put to good use. The ''Advice'' sections do not contain explanations. Instead, each piece of advice is accompanied by a reference to the appropriate section of the book. Where negative advice is given, that section usually provides a suggested alternative.

1.8.1 References

There are few direct references in the text, but here is a short list of books and papers that are mentioned directly or indirectly.

[Barton,1994] John J. Barton and Lee R. Nackman: *Scientific and Engineering C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-53393-6.

[Berg,1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project*. CACM. Vol. 38 No. 10. October 1995.

[Booch,1994] Grady Booch: *Object-Oriented Analysis and Design*. Benjamin/Cummings. Menlo Park, Calif. 1994. ISBN 0-8053-5340-2.

[Budge,1992] Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation using C++*. Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C,1990] X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association. Washington, DC, USA.

[C++,1998] X3 Secretariat: *International Standard – The C++ Language*. X3J16-14882. Information Technology Council (NSITC). Washington, DC, USA.

[Campbell,1987] Roy Campbell, et al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Coplien,1995] James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*. Addison-Wesley. Reading, Mass. 1995. ISBN 0-201-60734-4.

[Dahl,1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language.* Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dahl,1972] O-J. Dahl and C. A. R. Hoare: *Hierarchical Program Construction* in *Structured Programming.* Academic Press, New York. 1972.

[Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

[Gamma,1995] Erich Gamma, et al.: *Design Patterns*. Addison-Wesley. Reading, Mass. 1995. ISBN 0-201-63361-2.

[Goldberg,1983] A. Goldberg and D. Robson: *SMALLTALK-80 – The Language and Its Implementation.* Addison-Wesley. Reading, Mass. 1983.

[Griswold,1970] R. E. Griswold, et al.: *The Snobol4 Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1970.

[Griswold,1983] R. E. Griswold and M. T. Griswold: *The ICON Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1983.

[Hamilton,1993] G. Hamilton and P. Kougiouris: *The Spring Nucleus: A Microkernel for Objects*. Proc. 1993 Summer USENIX Conference. USENIX.

[Henricson,1997] Mats Henricson and Erik Nyquist: *Industrial Strength C++: Rules and Recommendations*. Prentice-Hall. Englewood Cliffs, New Jersey. 1997. ISBN 0-13-120965-5.

[Ichbiah,1979] Jean D. Ichbiah, et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14 No. 6. June 1979.

[Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology*. AT&T Technical Journal. Vol. 72 No. 5. September/October 1993.

[Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language (Second Edition).* Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. Vol. 1 No. 7. July 1989.

[Koenig,1997] Andrew Koenig and Barbara Moo: *Ruminations on C++*. Addison Wesley Longman. Reading, Mass. 1997. ISBN 0-201-42339-1.

[Knuth,1968] Donald Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Mass.

[Liskov,1979] Barbara Liskov et al.: *Clu Reference Manual*. MIT/LCS/TR-225. MIT Cambridge. Mass. 1979.

[Martin,1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall. Englewood Cliffs, New Jersey. 1995. ISBN 0-13-203837-4.

[Orwell,1949] George Orwell: *1984.* Secker and Warburg. London. 1949.

[Parrington,1995] Graham Parrington et al.: *The Design and Implementation of Arjuna*. Computer Systems. Vol. 8 No. 3. Summer 1995.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – The Language and Its Compiler*. Cambridge University Press, Cambridge. England. 1980. ISBN 0-521-21965-5.

[Rosler,1984] L. Rosler: *The Evolution of C – Past and Future.* AT&T Bell Laboratories Technical Journal. Vol. 63 No. 8. Part 2. October 1984.

[Rozier,1988] M. Rozier, et al.: *CHORUS Distributed Operating Systems*. Computing Systems. Vol. 1 No. 4. Fall 1988.

[Sethi,1981] Ravi Sethi: *Uniform Syntax for Type Expressions and Declarations*. Software Practice & Experience. Vol. 11. 1981.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). August, 1994.

[Stroustrup,1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Mass. 1986. ISBN 0-201-12078-X.

[Stroustrup,1987] Bjarne Stroustrup and Jonathan Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Mass. 1991. ISBN 0-201-53992-6.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-54330-3.

[Tarjan,1983] Robert E. Tarjan: *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics. Philadelphia, Penn. 1983. ISBN 0-898-71187-8.

[Unicode,1996] The Unicode Consortium: *The Unicode Standard, Version 2.0*. Addison-Wesley Developers Press. Reading, Mass. 1996. ISBN 0-201-48345-9.

[UNIX,1985] *UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition.* AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

[Wilson,1996] Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++*. The MIT Press. Cambridge. Mass. 1996. ISBN 0-262-73118-5.

[Wikström,1987] Åke Wikström: *Functional Programming Using ML*. Prentice-Hall. Englewood Cliffs, New Jersey. 1987.

[Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office. London. England. 1974.

References to books relating to design and larger software development issues can be found at the end of Chapter 23.