# C++ Solutions

## Companion to *The C++ Programming Language, Third Edition*

David Vandevoorde

**ADDISON–WESLEY**
**An Imprint of Addison Wesley Longman, Inc.**

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Computer and Engineering Publishing Group
Addison Wesley Longman, Inc.
One Jacob Way
Reading, Massachusetts 01867

*Library of Congress Cataloging-in-Publication Data*

Vandevoorde, David.
C++ solutions : companion to The C++ programming language, Third edition / David Vandevoorde.
p. cm.
Includes index.
ISBN 0-201-30965-3
1. C++ (Computer program language) I. Stroustrup, Bjarne. C++ programming language. II. Title.
QA76.73.C153V36 1998
005.13'3—DC21 98–20523
CIP



ISBN 0-201-30965-3
1 2 3 4 5 6 7 8 9—MA—0201009998
*First printing, July 1998*

*To Franklin T. Luk,*
*friend and mentor*

[ blank page ]

# Contents

# Acknowledgments

## Thank You

This book owes its existence to the contributions of many people.

It all started when Deborah Lafferty (an editor at Addison-Wesley) and Bjarne Stroustrup contacted me to write this book—I owe both of them for their initiative and for the suggestions they offered throughout the writing process. Marina Lang and Mike Hendrickson were also instrumental in the book design, and I must thank Mike for demonstrating useful Adobe FrameMaker procedures.

Or perhaps it all started when I bought my first copy of a book by Bjarne Stroustrup, or even when he decided to design a language that truly enhances my enjoyment of programming. Thus, I must thank those people who directly or indirectly taught me elements of C++: Bjarne Stroustrup, Josee Lajoie, Bill Gibbons, John Spicer, and Steve Adamczyk, in particular. Much of what is written in this book I also learned from discussions in the various technical C++ Usenet forums. This learning process continued through the many hours of voluntary work put in by the moderators of `comp.lang.c++.moderated` and `comp.std.c++`: Matt Austern, Stephen Clamage, Domenic De Vitto, Howard Harkness, Fergus Henderson, James Kanze, Dietmar Kühl, Robert Martin, John Potter, and Herb Sutter.

Eva Wong put up with my many hours of typing and grumbling around our home. Also many thanks to my parents and parents-in-law for their continuous encouragement.

Michael Beckmann, Sassan Hazeghi, and Chris Maitz of Hewlett-Packard were most helpful and supportive as I combined this project with my other professional responsibilities.

Franklin Luk deserves credit for his patience in teaching me—among other things—to write technical texts. I hope his efforts weren't wasted.

This book benefitted *immensely* from the corrections and suggestions by early reviewers: David Francis, Jeffrey Oldham, Srikanth Sankaran, Ed Schiebel, Jay Shain, Bjarne Stroustrup, Clovis Tondo, and Chris Van Wyk. Their contributions cannot be overstated.

*David Vandevoorde*
*Belmont, California*

[ blank page ]

# Chapter 1

# Introduction

Regardless of which programming languages are selected to write practical, effective programs, developing software is never trivial and often complex. Bjarne Stroustrup's *The C++ Programming Language (Third Edition)* shows how C++ provides a number of tools that can be applied to divide and conquer the challenges faced by modern software designers.

This conquest does requires a certain amount of experience, which in turn requires practice. No book can "supply practice," but Stroustrup provides a large number of exercises that can be used to review the knowledge and acquire experience. This book provides sample answers and discussions for a selection of these exercises.

## Overall Organization

The exercises selected for discussion in this book generally have the following three properties in common:

1. They illustrate important concepts or common uses and methods of C++.
2. Their solution is largely independent of a particular platform.
3. They can be solved fairly concisely.

I occasionally deviate from these rules when I feel the discussion warrants it. For example, a few exercises come with solutions that span several pages.

For easy reference, I follow Bjarne Stroustrup's chapter organization—for example, you will find the discussions about exercises presented in Chapter 15 of *The C++ Programming Language* in Chapter 15 of this book.

To accommodate the organization of this book, I had to decide what to do for the second and third chapters. In *The C++ Programming Language,* these chapters take the reader on a tour of the language and the library, but they do not present exercises. I chose to use these two chapters to provide a short presentation of fundamental C++ concepts and some

consequences of language evolution. Chapters 23 and 24 do not present exercises either; we used placeholders in this book.

This book is not a complete tutorial or a reference. Instead, the discussions frequently refer back to the third edition of *The C++ Programming Language*. References to specific paragraphs will use the § symbol. For instance, §C.3.1 refers to paragraph C.3.1 in appendix C of Stroustrup's book. However, I often elaborate on less-common C++ language constructs and even on common ones when they may result in subtle situations.

Almost every exercise statement is followed by a short Hints paragraph. These paragraphs point to related exercises, indicate relevant material in *The C++ Programming Language*, and suggest how fundamental or advanced the topic covered by the exercise is.

Most exercises can be solved in many ways. When no method appears clearly superior, I often suggest various alternative approaches. Even when I do think a particular solution is technically superior, I may still introduce a simpler one for the sake of illustration. Many problems—even the simpler ones—can easily lead to tangential experiments and discussions. I therefore sometimes point in these tangential directions using "supplementary exercises," which are new exercises not proposed in Stroustrup's book but which can be used to further explore various software development challenges. Note, however, that these exercises may involve solutions that are well beyond the scope of the answers in this book.

## Short Guide to the Exercises

The exercises in *The C++ Programming Language* were crafted as a vehicle to expand one's understanding of the possibilities of the language. Together, they cover a broad spectrum of C++ programming knowledge and solution components. Yet I still find that most of the selected exercises fall in one or a few of the following categories:

- *Facts and constructs:* Exercises that could be used as tests of one's familiarity with the language specifications. The following belong to this category: 4.2, 4.3, 4.7, 5.1, 5.5, 5.6, 6.1, 6.2, 6.4, 6.5, 6.6, 6.7, 6.9, 7.1, 7.2, 7.14, 8.4, 8.5, 8.7, 9.9, 10.1, 10.4, 11.1, 11.7, 12.1, 13.1, 13.15, 15.2, 16.15, 20.1.
- *Common idioms and good practices:* Illustrations of how C++ is often used to address various small programming problems. See Exercises 5.3, 5.4, 5.7, 5.10, 5.11, 5.12, 5.13, 6.3, 6.10, 6.11, 6.12, 6.13, 6.14, 6.16, 6.17, 6.19, 6.22, 7.3, 7.4, 7.6, 7.7, 7.9, 7.11, 7.16, 7.18, 7.19, 8.1, 8.10, 9.5, 10.2, 10.5, 10.12, 10.19, 11.3, 11.4, 11.8, 14.1, 14.2, 14.9, 15.3, 16.1, 16.2, 16.3, 16.6, 17.3, 17.6, 18.2, 18.5, 18.10, 18.11, 18.14, 19.1, 19.2, 19.3, 20.5, 20.16, 21.1, 21.2, 21.12, 21.18, 21.24, 22.1, 22.2, 22.4, 25.1, 25.5, 25.7, 25.19.
- *Experiments:* Exercises that include programs that illustrate implementation-dependent behavior or performance. Several exercises contain little benchmarks that are very useful in acquiring some intuition about the cost of various C++ features and techniques. See Exercises 4.3, 4.4, 4.5, 4.6, 5.8, 6.8, 8.6, 8.8, 9.1, 17.1, 18.19, 20.15.
- *Advanced idioms and design techniques:* Discussions of more specialized problems. Usually, I try to emphasize the general thought process leading to a solution in these

cases. Applying the resulting ideas to your own situation will often require some non-trivial adaptation. See Exercises 7.19, 12.10, 13.2, 13.16, 14.10, 21.13, 22.8.

- *Trivia and horrors:* Every programming language has its own little peculiarities and can be abused to produce unintelligible programs. A few exercises explore this darker side of software development, as well as pecularities that are not harmful but not fundamental either, as in Exercises 4.6, 5.9, 6.15, 10.15, 11.10, 11.20, 11.21, 12.9.

The fact that most exercises fall into one of the first two categories testifies to the fact that this book focuses primarily on C++ fundamentals. Once that is established, however, the more advanced material should become more accessible and lead to added productivity. In contrast, the exercises in the category *Experiments* do not usually require familiarity with the advanced C++ features.

## Suggestions

Don't feel compelled to work through the exercises in their numeric order. However, it may be useful to start with the exercises in the first category to refresh your fundamentals and then proceed with those in the second category (*Common idioms and good practices*) to get fluent with the language elements presented in the corresponding chapter. As is the case with *The C++ Programming Language,* the index in this book provides quick and easy access to topics of interest.

It is not always necessary to get completely familiar with the material in a chapter of *The C++ Programming Language* before trying out the exercises. However, it is useful to get a bird's eye view of the language by reading the second and third chapters of Stroustrup's book (the "Tour" chapters).

Of special interest are the slightly longer exercises. Exercise 15.3 is a complete object-oriented design and implementation of a popular board game. For an example implementation of standard-compliant elements, see 18.2 for an algorithm and 19.3 for a complete standard iterator adaptor. Exercise 20.5 is shorter but very useful for this purpose, as well. For a demonstration of the power of the standard libary, consider Exercise 6.12, which computes a variety of statistics on a set of data. If you're looking for an exercise to sharpen your skills with the more traditional features (statements and expressions), try 6.22.

I believe that to get the most out of this book, it is best to first try the exercises without reading the proposed solutions. Once you have found a solution to an exercise, you can compare your solution to the one in this book. In some cases, this comparison may suggest that there is really one "natural answer" to the exercise. But more likely, you will find that there are many ways to accomplish the same task. If a solution cannot be found within a reasonable time, reading the discussion of the corresponding exercise in this book will provide clarification. Often, this discussion will lead you to consider alternatives to those I propose, which is valuable practice.

Many exercises are best solved by actually implementing their solutions. There is much to be gained by sketching a solution and convincing yourself that the approach taken is valid. However, the actual implementation requires an additional level of attention to detail, and

good software design skills are acquired by cultivating a healthy respect for details. This is doubly true for the exercises in the *Experiments* category; do not hesitate to modify these experiments to further your insights into what makes C++ program tick and tick well.

Of course, good judgement is also needed to discriminate between the fundamental mechanisms and the details of the scaffolding. In some cases, you may find that one of the more annoying aspects of implementing solutions to the proposed exercises stems from having a C++ implementation that does not strictly comply with the ISO definition of the language. Chapter 3 deals with the most common issues encountered in this respect.

*The C++ Programming Language (Third Edition)* is, of course, highly recommended as a source of information to study the proposed software solutions. Once you are familiar with most techniques, you might want to sink your teeth into more literature. There are many excellent books on C++ out there that present material beyond the scope of *The C++ Programming Language*. I want to emphasize only a few:

*Effective C++: 50 Specific Ways to Improve Your Programs and Designs (Second Edition).* Scott Meyers. Reading, MA: Addison Wesley Longman, 1997.

*More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Scott Meyers. Reading, MA: Addison Wesley Longman, 1996.

*The Design and Evolution of C++.* Bjarne Stroustrup. Reading, MA: Addison Wesley Longman, 1994.

The first two books describe a large collection of rules of thumb and techniques that will lead to more robust and often more efficient C++ programs. General design principles and specific coding techniques are covered.

The third book describes the evolution of the C++ language up until about the end of 1993. I have found that understanding this evolution helps in determining how the rich set of C++ constructs can be best used. It also helps to make some of the subtler properties of C++ more intuitive and manageable.

Finally, I would like to point out that the C++ community is large. Discussion of techniques, semantics, and design principles with friends and colleagues is invaluable. In addition, there are many conferences and electronic forums that pertain to C++ programming. In particular, I have found that the Usenet forums `comp.lang.c++.moderated` and `comp.std.c++` provide enlightening discussions on a large variety of C++ issues.

# Chapter 2

# C++ Concepts

This book is not intended as a stand-alone tutorial for the C++ programming language. Instead, *The C++ Programming Language* is an excellent resource for that purpose. However, it is always useful and convenient to summarize some fundamental concepts and agree on terminology.

You can safely skip this chapter and return to it when you need a quick review of some fundamental C++ topic. For more complete and in-depth information refer to Stroustrup's book.

## Language versus Implementation

A programming language is an abstract thing. It is a set of rules and conventions to describe the behavior of programs. The concretization of this is a *language implementation*—a device, often a program itself, that translates a program description into actual behavior. The translation process always enforces some of the rules set by the programming language.

For C++, the implementation consists of the vast majority of cases of a *compiler*, which is a program that translates text (*source code*) into an *executable program*. Once translated, the program can then be run repeatedly on a *target platform* (a computer).

This book assumes that these concepts are not totally foreign to you. Some practical considerations are also discussed in Exercise 4.1.

## Tokens

C++ source code is usually divided up in multiple files (§9.1). When such a file is *preprocessed*, comments are stripped and macros are substituted. The result is a so-called *translation unit*. A translation unit is a sequence of *tokens*: words, numbers, and symbols. They are

the smallest entities that can have a meaning by themselves in C++. For example, the following snippet of C++ code:

```
int const hundred = 100 ;
```

consists of six tokens: '`int`', '`const`', '`hundred`', '`=`', '`100`', and '`;`'. *Identifiers* are tokens whose "spelling" can be freely chosen by the programmer. They consist of an arbitrary sequence of letters, digits, and underscores ('`_`'), but cannot start with a digit. Note that the capitalization of letters matters. Some sequences are, however, not allowed as identifiers:

- *Keywords* (§A.2) and *alternative representations* for certain symbols: '`int`' and '`const`' are examples of keywords. '`and`' is an example of an alternative token for the symbol '`&&`' (*logical and* operator). The C++ programming language has 63 keywords and 11 alternative representations.
- *Reserved names*: It is sometimes convenient for an implementation to introduce names of its own. Names containing two consecutive underscores (for instance, '`re__served`') and names starting with an underscore followed by a capital letter (for instance, '`_Reserved`') should, therefore, not be used as user-defined identifiers.

Tokens are often separated by *whitespace*, although that whitespace has no meaning of itself. In many cases, the transition from one token to the next does not require such whitespace and there are no restrictions on the amount of whitespace allowed between tokens. For example, the declaration above could appear in any of the following ways:

```
int const hundred=100;
int
   const
      hundred
         =
    100                                    ;
```

But the following would not work:

```
intconst hundred = 100;
```

Without intervening whitespace, `intconst` is a single—presumably user-defined—identifier rather than two keywords.

Certain uses of whitespace are detrimental to source code clarity. Our third example illustrates this. Nonetheless, individual tastes vary when it comes to source code style, and it would be a futile exercise for this book to try to impose a set of rules in this regard. Many projects do follow certain guidelines for formatting C++ source code in an attempt to ease the understanding of source files (see also §4.9.3 and Exercise 6.23).

## Names, Declarations, and Scopes

Many entities in C++ are referred to by using a *name*. The introduction or *re*introduction of such a name is called a *declaration*. The part of a translation unit in which a declaration can

be referred to by its plain associated name is called the *scope* of that declaration. For example:

```
int a = 3;
int b = 2; // <- scope of global b starts here

void f() {
   int b = 0; // <- scope of global b stops here,
   ++a;       //    scope of local b starts
   ++b; // Refers to local b
} // <- scope of local b ends, scope of global b resumes
```

As the example shows, two different declarations can be associated with a single name. When that happens, the "innermost" name hides the outer one. In some cases, hidden names and names in nonactive scopes can be accessed by using so-called *qualified names*—names containing the *scope-resolution operator* '`::`'. For example:

```
int a;

namespace N {
   int a, b;
   void f() {
      ::a = 1; // Qualified name for (hidden) global 'a'
      a = 10; // Unqualified name accesses inner 'a'
    }
}

void g() {
   a = 10; // Global 'a'
   N::a = 7; // 'a' in namespace scope (qualified name)
}
```

A *qualifier* on the left of the scope-resolution operator must be either the name of a namespace (§8.2) or that of a class (§10.2). This implies that names local to a function cannot be accessed through a qualified name.

## Objects, Types, References, and Functions

Objects are things that occupy a contiguous region of storage (computer memory). However, a random set of contiguous bits of information is usually not considered an object. Instead, an object has a *type* associated with its physical bits. A type is thus a collection of attributes and applicable operations that determine how the bits constituting an object must be interpreted. For example, on many systems the definitions

```
char c = 1;
```

and

```
bool b = true;
```

result in exactly the same set of physical binary values. Yet because their types are different, ++c and ++b will most likely have different physical values.

C++ has a number of built-in *fundamental types*; `bool`, `char`, `unsigned int`, and `double` are examples of these (§4.2, §4.3, §4.4, §4.5). Additional types are called *compound types* and can be created in a variety of ways. These compound types fall into one of the following categories:

- Enumerations (§4.8)
- Classes (including structures; §5.7, §10)
- Unions (§10.4.12)
- Arrays (§5.2)
- Pointers (§5.1, §5.3)
- References (§5.5)
- Functions (§7)

Although functions and references have types, they are not *objects* in the C++ sense of the word. References are "aliases" for objects—as such, they do not always require storage at run-time; instead, the alias can sometimes be substituted by the C++ translator. Functions typically do require run-time storage for their representation, but C++—like most other high-level programming languages—does not make that storage available to the programmer. This prevents us, for example, from writing so-called self-modifying code.

## Lvalue and Rvalue Expressions

Objects can exist because you explicitly define them or allocate them (for example, by using the `new` operation). If an expression refers to such an object, the expression is called an *lvalue*. Otherwise, it is called an *rvalue*. For example:

```
int a;
int *b = &a;
a = 2; // 'a' is an lvalue, '2' is an rvalue
*b = a; // Both '*b' and 'a' are lvalues
b = new int; // 'new int' is an rvalue; 'b' is an lvalue
int const &ri = *new int; // '*new int' is an lvalue
a = ri; // 'ri' is an lvalue
int f();
*b = f(); // 'f()' is an rvalue
int& g();
a = g(); // 'g()' is an lvalue
```

The *l* in the term *lvalue* was inherited from ancestors of the C language (for example, BCPL, §1) in which lvalue could appear on the *left* side of an assignment, whereas rvalues could not. In modern C and in C++, lvalues may not always appear on the left of an assignment operator because the referenced object is `const`. Such lvalues are called *nonmodifiable lvalues*. C++ also provides ways to modify rvalues, but it is rarely necessary to do this.

The results of arithmetic operators (+, -, %, *, ...) and the results of function calls are often rvalues unless they return references.

## Initialization versus Assignment

*Initialization* (§4.9.5) is the process that creates the first value of an object. *Assignment* (§2.3.1), on the other hand, is an operation that usually changes the value of an object—a previous value existed. Assignment is expressed using the token '=,' but sometimes initialization can also be indicated with that symbol:

```
int a = 12; // Initialization (definition)
a = 13; // Assignment
```

Basically, the '=' symbol represents initialization when it appears as part of a declaration; otherwise, it is assignment.

Initialization can also occur without the '=' token, as well. For example:

```
int a(12); // Same as above
```

Also, the transfer of function arguments happens through initialization. For example,

```
void f(X x) { /* ... */ }

void g() {
   X a; // Assume X is some type
   f(a); // parameter 'x' acquires 'a' value
}        // by initialization
```

For classes, you can define your own initialization procedure by providing constructors (§10.2.3, §10.4) and your own assignment operation by providing an `operator=` (§10.4.4.1). Remember that your `operator=` *will not* be called for initialization even if that initialization is expressed using the '=' symbol.

## Declaration Syntax

The syntax of the most commonly occurring declarations is relatively intuitive, but the general syntax rules can sometimes be surprising. The discussion below summarizes some of the fundamentals explained in §5, §6, §7, and §10.

### Variables and Constants

A declaration consists of four parts (§4.9.1):

1. A set of "specifiers" (optional)
2. A base type
3. A declarator
4. An optional initializer

The specifiers correspond to a variety of properties associated with the entity being declared—for example, `extern` (§9.2), `virtual` (§12.2.6), and `explicit` (§11.7.1)—or with the type of that entity (for example, `const`, §5.4). Interestingly, there are few limitations to the order in which specifiers appear, and, in fact, they may even appear after the base type. For example, all of the following are legal and equivalent:

```
extern int const c;
const int extern c;
const extern int c;
int extern const c;
```

Almost all programmers prefer to keep the specifiers associated with the type (the *type specifiers*) close to the base type; the other specifiers usually precede the base type and the type specifiers. The above declaration is thus generally ordered in one of the following ways:

```
extern const int c; // "const" is a type-specifier,
extern int const c; // "extern" is not
```

The former ordering has been the most popular, but the latter is sometimes preferred for reasons explained in Exercise 5.1.

The *base type* can be a fundamental type (for example, `float`) or the name of an `enum`, a `class` (or `struct`), or a `union`. If the base type is not a fundamental type, its name can be qualified (see above) and *elaborated*. The latter means that the—possibly qualified—name is preceded by one of the keywords `enum`, `struct`, `class`, or `union`. For example:

```
namespace N {
   enum E { two = 2, seven = 7 };
   E a; // Name E is not qualified and not elaborated
   enum E b; // Elaborated, but not qualified name E
}
enum N::E c; // Elaborated and qualified name E
```

The *declarator* part of a declaration consists of the declared name optionally surrounded by *declarator operators* that modify the base type with which the name is being associated. These operators are used to create references, pointers, arrays, functions, and combinations of these things. Let's first concentrate on pointers, which are declared using a (prefix) asterisk '`*`.' A pointer `p` to an integer can thus be declared as follows:

```
int *p;
```