BJARNE STROUSTRUP

THE CREATOR OF C++

Using
C++11
and
C++14

PROGRAMMING

Principles and Practice Using C++

SECOND EDITION

FREE SAMPLE CHAPTER

Programming

Second Edition

This page intentionally left blank

Programming

Principles and Practice Using C++

Second Edition

Bjarne Stroustrup

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

A complete list of photo sources and credits appears on pages 1273–1274.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Stroustrup, Bjarne, author.
Programming : principles and practice using C++ / Bjarne Stroustrup. – Second edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-99278-9 (pbk. : alk. paper)
1. C++ (Computer program language) I. Title.
QA76.73.C153S82 2014
005.13'3–dc23
2014004197

ISBN-13: 978-0-321-99278-9
ISBN-10: 0-321-99278-4
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, May 2014

Contents

This page intentionally left blank

Preface

> "Damn the torpedoes!
> Full speed ahead."
>
> **—Admiral Farragut**

Programming is the art of expressing solutions to problems so that a computer can execute those solutions. Much of the effort in programming is spent finding and refining solutions. Often, a problem is only fully understood through the process of programming a solution for it.

This book is for someone who has never programmed before but is willing to work hard to learn. It helps you understand the principles and acquire the practical skills of programming using the C++ programming language. My aim is for you to gain sufficient knowledge and experience to perform simple useful programming tasks using the best up-to-date techniques. How long will that take? As part of a first-year university course, you can work through this book in a semester (assuming that you have a workload of four courses of average difficulty). If you work by yourself, don't expect to spend less time than that (maybe 15 hours a week for 14 weeks).

Three months may seem a long time, but there's a lot to learn and you'll be writing your first simple programs after about an hour. Also, all learning is gradual: each chapter introduces new useful concepts and illustrates them with examples inspired by real-world uses. Your ability to express ideas in code – getting a computer to do what you want it to do – gradually and steadily increases as you go along. I never say, "Learn a month's worth of theory and then see if you can use it."

Why would you want to program? Our civilization runs on software. Without understanding software you are reduced to believing in "magic" and will be locked out of many of the most interesting, profitable, and socially useful technical fields of work. When I talk about programming, I think of the whole spectrum of computer programs from personal computer applications with GUIs (graphical user interfaces), through engineering calculations and embedded systems control applications (such as digital cameras, cars, and cell phones), to text manipulation applications as found in many humanities and business applications. Like mathematics, programming – when done well – is a valuable intellectual exercise that sharpens our ability to think. However, thanks to feedback from the computer, programming is more concrete than most forms of math, and therefore accessible to more people. It is a way to reach out and change the world – ideally for the better. Finally, programming can be great fun.

Why C++? You can't learn to program without a programming language, and C++ directly supports the key concepts and techniques used in real-world software. C++ is one of the most widely used programming languages, found in an unsurpassed range of application areas. You find C++ applications everywhere from the bottom of the oceans to the surface of Mars. C++ is precisely and comprehensively defined by a nonproprietary international standard. Quality and/or free implementations are available on every kind of computer. Most of the programming concepts that you will learn using C++ can be used directly in other languages, such as C, C#, Fortran, and Java. Finally, I simply like C++ as a language for writing elegant and efficient code.

This is not the easiest book on beginning programming; it is not meant to be. I just aim for it to be the easiest book from which you can learn the basics of real-world programming. That's quite an ambitious goal because much modern software relies on techniques considered advanced just a few years ago.

My fundamental assumption is that you want to write programs for the use of others, and to do so responsibly, providing a decent level of system quality; that is, I assume that you want to achieve a level of professionalism. Consequently, I chose the topics for this book to cover what is needed to get started with real-world programming, not just what is easy to teach and learn. If you need a technique to get basic work done right, I describe it, demonstrate concepts and language facilities needed to support the technique, provide exercises for it, and expect you to work on those exercises. If you just want to understand toy programs, you can get along with far less than I present. On the other hand, I won't waste your time with material of marginal practical importance. If an idea is explained here, it's because you'll almost certainly need it.

If your desire is to use the work of others without understanding how things are done and without adding significantly to the code yourself, this book is not for you. If so, please consider whether you would be better served by another book and another language. If that is approximately your view of programming, please also consider from where you got that view and whether it in fact is adequate for your needs. People often underestimate the complexity of programming as well as its value. I would hate for you to acquire a dislike for programming because of a mismatch between what you need and the part of the software reality I describe. There are many parts of the "information technology" world that do not require knowledge of programming. This book is aimed to serve those who do want to write or understand nontrivial programs.

Because of its structure and practical aims, this book can also be used as a second book on programming for someone who already knows a bit of C++ or for someone who programs in another language and wants to learn C++. If you fit into one of those categories, I refrain from guessing how long it will take you to read this book, but I do encourage you to do many of the exercises. This will help you to counteract the common problem of writing programs in older, familiar styles rather than adopting newer techniques where these are more appropriate. If you have learned C++ in one of the more traditional ways, you'll find something surprising and useful before you reach Chapter 7. Unless your name is Stroustrup, what I discuss here is not "your father's C++."

Programming is learned by writing programs. In this, programming is similar to other endeavors with a practical component. You cannot learn to swim, to play a musical instrument, or to drive a car just from reading a book – you must practice. Nor can you learn to program without reading and writing lots of code. This book focuses on code examples closely tied to explanatory text and diagrams. You need those to understand the ideals, concepts, and principles of programming and to master the language constructs used to express them. That's essential, but by itself, it will not give you the practical skills of programming. For that, you need to do the exercises and get used to the tools for writing, compiling, and running programs. You need to make your own mistakes and learn to correct them. There is no substitute for writing code. Besides, that's where the fun is!

On the other hand, there is more to programming – much more – than following a few rules and reading the manual. This book is emphatically not focused on "the syntax of C++." Understanding the fundamental ideals, principles, and techniques is the essence of a good programmer. Only well-designed code has a chance of becoming part of a correct, reliable, and maintainable system. Also, "the fundamentals" are what last: they will still be essential after today's languages and tools have evolved or been replaced.

What about computer science, software engineering, information technology, etc.? Is that all programming? Of course not! Programming is one of the fundamental topics that underlie everything in computer-related fields, and it has a natural place in a balanced course of computer science. I provide brief introductions to key concepts and techniques of algorithms, data structures, user interfaces, data processing, and software engineering. However, this book is not a substitute for a thorough and balanced study of those topics.

Code can be beautiful as well as useful. This book is written to help you see that, to understand what it means for code to be beautiful, and to help you to master the principles and acquire the practical skills to create such code. Good luck with programming!

A note to students

Of the many thousands of first-year students we have taught so far using this book at Texas A&M University, about 60% had programmed before and about 40% had never seen a line of code in their lives. Most succeeded, so you can do it, too.

You don't have to read this book as part of a course. The book is widely used for self-study. However, whether you work your way through as part of a course or independently, try to work with others. Programming has an – unfair – reputation as a lonely activity. Most people work better and learn faster when they are part of a group with a common aim. Learning together and discussing problems with friends is not cheating! It is the most efficient – as well as most pleasant – way of making progress. If nothing else, working with friends forces you to articulate your ideas, which is just about the most efficient way of testing your understanding and making sure you remember. You don't actually have to personally discover the answer to every obscure language and programming environment problem. However, please don't cheat yourself by not doing the drills and a fair number of exercises (even if no teacher forces you to do them). Remember: programming is (among other things) a practical skill that you need to practice to master. If you don't write code (do several exercises for each chapter), reading this book will be a pointless theoretical exercise.

Most students – especially thoughtful good students – face times when they wonder whether their hard work is worthwhile. When (not if) this happens to you, take a break, reread this Preface, and look at Chapter 1 ("Computers, People, and Programming") and Chapter 22 ("Ideals and History"). There, I try to articulate what I find exciting about programming and why I consider it a crucial tool for making a positive contribution to the world. If you wonder about my teaching philosophy and general approach, have a look at Chapter 0 ("Notes to the Reader").

You might find the weight of this book worrying, but it should reassure you that part of the reason for the heft is that I prefer to repeat an explanation or add an example rather than have you search for the one and only explanation. The other major reason is that the second half of the book is reference material and "additional material" presented for you to explore only if you are interested in more information about a specific area of programming, such as embedded systems programming, text analysis, or numerical computation.

And please don't be too impatient. Learning any major new and valuable skill takes time and is worth it.

A note to teachers

No. This is not a traditional Computer Science 101 course. It is a book about how to construct working software. As such, it leaves out much of what a computer science student is traditionally exposed to (Turing completeness, state machines, discrete math, Chomsky grammars, etc.). Even hardware is ignored on the assumption that students have used computers in various ways since kindergarten. This book does not even try to mention most important CS topics. It is about programming (or more generally about how to develop software), and as such it goes into more detail about fewer topics than many traditional courses. It tries to do just one thing well, and computer science is not a one-course topic. If this book/course is used as part of a computer science, computer engineering, electrical engineering (many of our first students were EE majors), information science, or whatever program, I expect it to be taught alongside other courses as part of a well-rounded introduction.

Please read Chapter 0 ("Notes to the Reader") for an explanation of my teaching philosophy, general approach, etc. Please try to convey those ideas to your students along the way.

ISO standard C++

C++ is defined by an ISO standard. The first ISO C++ standard was ratified in 1998, so that version of C++ is known as C++98. I wrote the first edition of this book while working on the design of C++11. It was most frustrating not to be able to use the novel features (such as uniform initialization, range-**for**-loops, move semantics, lambdas, and concepts) to simplify the presentation of principles and techniques. However, the book was designed with C++11 in mind, so it was relatively easy to "drop in" the features in the contexts where they belonged. As of this writing, the current standard is C++11 from 2011, and facilities from the upcoming 2014 ISO standard, C++14, are finding their way into mainstream C++ implementations. The language used in this book is C++11 with a few C++14 features. For example, if your compiler complains about

```
vector<int> v1;
vector<int> v2 {v1};    // C++14-style copy construction
```

use

```
vector<int> v1;
vector<int> v2 = v1;    // C++98-style copy construction
```

instead.

If your compiler does not support C++11, get a new compiler. Good, modern C++ compilers can be downloaded from a variety of suppliers; see www.stroustrup.com/compilers.html. Learning to program using an earlier and less supportive version of the language can be unnecessarily hard.

Support

The book's support website, www.stroustrup.com/Programming, contains a variety of material supporting the teaching and learning of programming using this book. The material is likely to be improved with time, but for starters, you can find

- Slides for lectures based on the book
- An instructor's guide
- Header files and implementations of libraries used in the book
- Code for examples in the book
- Solutions to selected exercises
- Potentially useful links
- Errata

Suggestions for improvements are always welcome.

Acknowledgments

I'd especially like to thank my late colleague and co-teacher Lawrence "Pete" Petersen for encouraging me to tackle the task of teaching beginners long before I'd otherwise have felt comfortable doing that, and for supplying the practical teaching experience to make the course succeed. Without him, the first version of the course would have been a failure. We worked together on the first versions of the course for which this book was designed and together taught it repeatedly, learning from our experiences, improving the course and the book. My use of "we" in this book initially meant "Pete and me."

Thanks to the students, teaching assistants, and peer teachers of ENGR 112, ENGR 113, and CSCE 121 at Texas A&M University who directly and indirectly helped us construct this book, and to Walter Daugherity, Hyunyoung Lee, Teresa Leyk, Ronnie Ward, and Jennifer Welch, who have also taught the course. Also thanks to Damian Dechev, Tracy Hammond, Arne Tolstrup Madsen, Gabriel Dos Reis, Nicholas Stroustrup, J. C. van Winkel, Greg Versoonder, Ronnie Ward, and Leor Zolman for constructive comments on drafts of this book. Thanks to Mogens Hansen for explaining about engine control software. Thanks to Al Aho, Stephen Edwards, Brian Kernighan, and Daisy Nguyen for helping me hide away from distractions to get writing done during the summers.

Thanks to Art Werschulz for many constructive comments based on his use of the first edition of this book in courses at Fordham University in New York City and to Nick Maclaren for many detailed comments on the exercises based on his use of the first edition of this book at Cambridge University. His students had dramatically different backgrounds and professional needs from the TAMU first-year students.

Thanks to the reviewers that Addison-Wesley found for me. Their comments, mostly based on teaching either C++ or Computer Science 101 at the college level, have been invaluable: Richard Enbody, David Gustafson, Ron McCarty, and K. Narayanaswamy. Also thanks to my editor, Peter Gordon, for many useful comments and (not least) for his patience. I'm very grateful to the production team assembled by Addison-Wesley; they added much to the quality of the book: Linda Begley (proofreader), Kim Arney (compositor), Rob Mauhar (illustrator), Julie Nahil (production editor), and Barbara Wood (copy editor).

Thanks to the translators of the first edition, who found many problems and helped clarify many points. In particular, Loïc Joly and Michel Michaud did a thorough technical review of the French translation that led to many improvements.

I would also like to thank Brian Kernighan and Doug McIlroy for setting a very high standard for writing about programming, and Dennis Ritchie and Kristen Nygaard for providing valuable lessons in practical language design.

This page intentionally left blank

Notes to the Reader

> "When the terrain disagrees with the map, trust the terrain."
>
> **—Swiss army proverb**

This chapter is a grab bag of information; it aims to give you an idea of what to expect from the rest of the book. Please skim through it and read what you find interesting. A teacher will find most parts immediately useful. If you are reading this book without the benefit of a good teacher, please don't try to read and understand everything in this chapter; just look at "The structure of this book" and the first part of the "A philosophy of teaching and learning" sections. You may want to return and reread this chapter once you feel comfortable writing and executing small programs.

0.1 The structure of this book

This book consists of four parts and a collection of appendices:

- *Part I, "The Basics,"* presents the fundamental concepts and techniques of programming together with the C++ language and library facilities needed to get started writing code. This includes the type system, arithmetic operations, control structures, error handling, and the design, implementation, and use of functions and user-defined types.
- *Part II, "Input and Output,"* describes how to get numeric and text data from the keyboard and from files, and how to produce corresponding output to the screen and to files. Then, it shows how to present numeric data, text, and geometric shapes as graphical output, and how to get input into a program from a graphical user interface (GUI).
- *Part III, "Data and Algorithms,"* focuses on the C++ standard library's containers and algorithms framework (the STL, standard template library). It shows how containers (such as **vector**, **list**, and **map**) are implemented (using pointers, arrays, dynamic memory, exceptions, and templates) and used. It also demonstrates the design and use of standard library algorithms (such as **sort**, **find**, and **inner_product**).
- *Part IV, "Broadening the View,"* offers a perspective on programming through a discussion of ideals and history, through examples (such as matrix computation, text manipulation, testing, and embedded systems programming), and through a brief description of the C language.
- *Appendices* provide useful information that doesn't fit into a tutorial presentation, such as surveys of C++ language and standard library facilities, and descriptions of how to get started with an integrated development environment (IDE) and a graphical user interface (GUI) library.

Unfortunately, the world of programming doesn't really fall into four cleanly separated parts. Therefore, the "parts" of this book provide only a coarse classification of topics. We consider it a useful classification (obviously, or we wouldn't have used it), but reality has a way of escaping neat classifications. For example, we need to use input operations far sooner than we can give a thorough explanation of C++ standard I/O streams (input/output streams). Where the set of topics needed to present an idea conflicts with the overall classification, we explain the minimum needed for a good presentation, rather than just referring to the complete explanation elsewhere. Rigid classifications work much better for manuals than for tutorials.

The order of topics is determined by programming techniques, rather than programming language features; see §0.2. For a presentation organized around language features, see Appendix A.

To ease review and to help you if you miss a key point during a first reading where you have yet to discover which kind of information is crucial, we place three kinds of "alert markers" in the margin:

- Blue: concepts and techniques (this paragraph is an example of that)
- Green: advice
- Red: warning

0.1.1 General approach

In this book, we address you directly. That is simpler and clearer than the conventional "professional" indirect form of address, as found in most scientific papers. By "you" we mean "you, the reader," and by "we" we refer either to "ourselves, the author and teachers," or to you and us working together through a problem, as we might have done had we been in the same room.

This book is designed to be read chapter by chapter from the beginning to the end. Often, you'll want to go back to look at something a second or a third time. In fact, that's the only sensible approach, as you'll always dash past some details that you don't yet see the point in. In such cases, you'll eventually go back again. However, despite the index and the cross-references, this is not a book that you can open to any page and start reading with any expectation of success. Each section and each chapter assume understanding of what came before.

Each chapter is a reasonably self-contained unit, meant to be read in "one sitting" (logically, if not always feasible on a student's tight schedule). That's one major criterion for separating the text into chapters. Other criteria include that a chapter is a suitable unit for drills and exercises and that each chapter presents some specific concept, idea, or technique. This plurality of criteria has left a few chapters uncomfortably long, so please don't take "in one sitting" too literally. In particular, once you have thought about the review questions, done the drill, and worked on a few exercises, you'll often find that you have to go back to reread a few sections and that several days have gone by. We have clustered the chapters into "parts" focused on a major topic, such as input/output. These parts make good units of review.

Common praise for a textbook is "It answered all my questions just as I thought of them!" That's an ideal for minor technical questions, and early readers have observed the phenomenon with this book. However, that cannot be the whole ideal. We raise questions that a novice would probably not think of. We aim to ask and answer questions that you need to consider when writing quality software for the use of others. Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer. Asking only the easy and obvious questions would make you feel good, but it wouldn't help make you a programmer.

We try to respect your intelligence and to be considerate about your time. In our presentation, we aim for professionalism rather than cuteness, and we'd rather understate a point than hype it. We try not to exaggerate the importance of a programming technique or a language feature, but please don't underestimate a simple statement like "This is often useful." If we quietly emphasize that something is important, we mean that you'll sooner or later waste days if you don't master it. Our use of humor is more limited than we would have preferred, but experience shows that people's ideas of what is funny differ dramatically and that a failed attempt at humor can be confusing.

We do not pretend that our ideas or the tools offered are perfect. No tool, library, language, or technique is "the solution" to all of the many challenges facing a programmer. At best, it can help you to develop and express your solution. We try hard to avoid "white lies"; that is, we refrain from oversimplified explanations that are clear and easy to understand, but not true in the context of real languages and real problems. On the other hand, this book is not a reference; for more precise and complete descriptions of C++, see Bjarne Stroustrup, *The C++ Programming Language, Fourth Edition* (Addison-Wesley, 2013), and the ISO C++ standard.

0.1.2 Drills, exercises, etc.

Programming is not just an intellectual activity, so writing programs is necessary to master programming skills. We provide two levels of programming practice:

- *Drills:* A drill is a very simple exercise devised to develop practical, almost mechanical skills. A drill usually consists of a sequence of modifications of a single program. You should do every drill. A drill is not asking for deep understanding, cleverness, or initiative. We consider the drills part of the basic fabric of the book. If you haven't done the drills, you have not "done" the book.

- *Exercises:* Some exercises are trivial and others are very hard, but most are intended to leave some scope for initiative and imagination. If you are serious, you'll do quite a few exercises. At least do enough to know which are difficult for you. Then do a few more of those. That's how you'll learn the most. The exercises are meant to be manageable without exceptional cleverness, rather than to be tricky puzzles. However, we hope that we have provided exercises that are hard enough to challenge anybody and enough exercises to exhaust even the best student's available time. We do not expect you to do them all, but feel free to try.

In addition, we recommend that you (every student) take part in a small project (and more if time allows for it). A project is intended to produce a complete useful program. Ideally, a project is done by a small group of people (e.g., three people) working together for about a month while working through the chapters in Part III. Most people find the projects the most fun and what ties everything together.

Some people like to put the book aside and try some examples before reading to the end of a chapter; others prefer to read ahead to the end before trying to get code to run. To support readers with the former preference, we provide simple suggestions for practical work labeled "**Try this**" at natural breaks in the text. A **Try this** is generally in the nature of a drill focused narrowly on the topic that precedes it. If you pass a **Try this** without trying – maybe because you are not near a computer or you find the text riveting – do return to it when you do the chapter drill; a **Try this** either complements the chapter drill or is a part of it.

At the end of each chapter you'll find a set of review questions. They are intended to point you to the key ideas explained in the chapter. One way to look at the review questions is as a complement to the exercises: the exercises focus on the practical aspects of programming, whereas the review questions try to help you articulate the ideas and concepts. In that, they resemble good interview questions.

The "Terms" section at the end of each chapter presents the basic vocabulary of programming and of C++. If you want to understand what people say about programming topics and to articulate your own ideas, you should know what each means.

Learning involves repetition. Our ideal is to make every important point at least twice and to reinforce it with exercises.

0.1.3 What comes after this book?

At the end of this book, will you be an expert at programming and at C++? Of course not! When done well, programming is a subtle, deep, and highly skilled art building on a variety of technical skills. You should no more expect to be an expert at programming in four months than you should expect to be an expert in biology, in math, in a natural language (such as Chinese, English, or Danish), or at playing the violin in four months – or in half a year, or a year. What you should hope for, and what you can expect if you approach this book seriously, is to have a really good start that allows you to write relatively simple useful programs, to be able to read more complex programs, and to have a good conceptual and practical background for further work.

The best follow-up to this initial course is to work on a real project developing code to be used by someone else. After that, or (even better) in parallel with a real project, read either a professional-level general textbook (such as Stroustrup, *The C++ Programming Language*), a more specialized book relating to the needs of your project (such as Qt for GUI, or ACE for distributed programming), or a textbook focusing on a particular aspect of C++ (such as Koenig and Moo, *Accelerated C++*; Sutter's *Exceptional C++*; or Gamma et al., *Design Patterns*). For more references, see §0.6 or the Bibliography section at the back of the book.

Eventually, you should learn another programming language. We don't consider it possible to be a professional in the realm of software – even if you are not primarily a programmer – without knowing more than one language.

0.2 A philosophy of teaching and learning

What are we trying to help you learn? And how are we approaching the process of teaching? We try to present the minimal concepts, techniques, and tools for you to do effective practical programs, including

- Program organization
- Debugging and testing
- Class design
- Computation
- Function and algorithm design
- Graphics (two-dimensional only)
- Graphical user interfaces (GUIs)
- Text manipulation
- Regular expression matching
- Files and stream input and output (I/O)
- Memory management
- Scientific/numerical/engineering calculations
- Design and programming ideals
- The C++ standard library
- Software development strategies
- C-language programming techniques

Working our way through these topics, we cover the programming techniques called procedural programming (as with the C programming language), data abstraction, object-oriented programming, and generic programming. The main topic of this book is *programming,* that is, the ideals, techniques, and tools of expressing ideas in code. The C++ programming language is our main tool, so we describe many of C++'s facilities in some detail. But please remember that C++ is just a tool, rather than the main topic of this book. This is "programming using C++," not "C++ with a bit of programming theory."

Each topic we address serves at least two purposes: it presents a technique, concept, or principle and also a practical language or library feature. For example, we use the interface to a two-dimensional graphics system to illustrate the use of classes and inheritance. This allows us to be economical with space (and your time) and also to emphasize that programming is more than simply slinging code together to get a result as quickly as possible. The C++ standard library is a major source of such "double duty" examples – many even do triple duty. For example, we introduce the standard library **vector**, use it to illustrate widely useful design techniques, and show many of the programming techniques used to implement it. One of our aims is to show you how major library facilities are implemented and how they map to hardware. We insist that craftsmen must understand their tools, not just consider them "magical."

Some topics will be of greater interest to some programmers than to others. However, we encourage you not to prejudge your needs (how would you know what you'll need in the future?) and at least look at every chapter. If you read this book as part of a course, your teacher will guide your selection.

We characterize our approach as "depth-first." It is also "concrete-first" and "concept-based." First, we quickly (well, relatively quickly, Chapters 1–11) assemble a set of skills needed for writing small practical programs. In doing so, we present a lot of tools and techniques in minimal detail. We focus on simple concrete code examples because people grasp the concrete faster than the abstract. That's simply the way most humans learn. At this initial stage, you should not expect to understand every little detail. In particular, you'll find that trying something slightly different from what just worked can have "mysterious" effects. Do try, though! And please do the drills and exercises we provide. Just remember that early on you just don't have the concepts and skills to accurately estimate what's simple and what's complicated; expect surprises and learn from them.

We move fast in this initial phase – we want to get you to the point where you can write interesting programs as fast as possible. Someone will argue, "We must move slowly and carefully; we must walk before we can run!" But have you ever watched a baby learning to walk? Babies really do run by themselves before they learn the finer skills of slow, controlled walking. Similarly, you will dash ahead, occasionally stumbling, to get a feel of programming before slowing down to gain the necessary finer control and understanding. You must run before you can walk!

It is essential that you don't get stuck in an attempt to learn "everything" about some language detail or technique. For example, you could memorize all of C++'s built-in types and all the rules for their use. Of course you could, and doing so might make you feel knowledgeable. However, it would not make you a programmer. Skipping details will get you "burned" occasionally for lack of knowledge, but it is the fastest way to gain the perspective needed to write good programs. Note that our approach is essentially the one used by children learning their native language and also the most effective approach used to teach foreign languages. We encourage you to seek help from teachers, friends, colleagues, instructors, Mentors, etc. on the inevitable occasions when you are stuck. Be assured that nothing in these early chapters is fundamentally difficult. However, much will be unfamiliar and might therefore feel difficult at first.

Later, we build on the initial skills to broaden your base of knowledge and skills. We use examples and exercises to solidify your understanding, and to provide a conceptual base for programming.

We place a heavy emphasis on ideals and reasons. You need ideals to guide you when you look for practical solutions – to know when a solution is good and principled. You need to understand the reasons behind those ideals to understand why they should be your ideals, why aiming for them will help you and the users of your code. Nobody should be satisfied with "because that's the way it is" as an explanation. More importantly, an understanding of ideals and reasons allows you to generalize from what you know to new situations and to combine ideas and tools in novel ways to address new problems. Knowing "why" is an essential part of acquiring programming skills. Conversely, just memorizing lots of poorly understood rules and language facilities is limiting, a source of errors, and a massive waste of time. We consider your time precious and try not to waste it.

Many C++ language-technical details are banished to appendices and manuals, where you can look them up when needed. We assume that you have the initiative to search out information when needed. Use the index and the table of contents. Don't forget the online help facilities of your compiler, and the web. Remember, though, to consider every web resource highly suspect until you have reason to believe better of it. Many an authoritative-looking website is put up by a programming novice or someone with something to sell. Others are simply outdated. We provide a collection of links and information on our support website: www.stroustrup.com/Programming.

Please don't be too impatient for "realistic" examples. Our ideal example is the shortest and simplest code that directly illustrates a language facility, a concept, or a technique. Most real-world examples are far messier than ours, yet do not consist of more than a combination of what we demonstrate. Successful commercial programs with hundreds of thousands of lines of code are based on techniques that we illustrate in a dozen 50-line programs. The fastest way to understand real-world code is through a good understanding of the fundamentals.

On the other hand, we do not use "cute examples involving cuddly animals" to illustrate our points. We assume that you aim to write real programs to be used by real people, so every example that is not presented as language-technical is taken from a real-world use. Our basic tone is that of professionals addressing (future) professionals.

0.2.1 The order of topics

There are many ways to teach people how to program. Clearly, we don't subscribe to the popular "the way I learned to program is the best way to learn" theories. To ease learning, we early on present topics that would have been considered advanced only a few years ago. Our ideal is for the topics we present to be driven by problems you meet as you learn to program, to flow smoothly from topic to topic as you increase your understanding and practical skills. The major flow of this book is more like a story than a dictionary or a hierarchical order.

It is impossible to learn all the principles, techniques, and language facilities needed to write a program at once. Consequently, we have to choose a subset of principles, techniques, and features to start with. More generally, a textbook or a course must lead students through a series of subsets. We consider it our responsibility to select topics and to provide emphasis. We can't just present everything, so we must choose; what we leave out is at least as important as what we leave in – at each stage of the journey.

For contrast, it may be useful for you to see a list of (severely abbreviated) characterizations of approaches that we decided not to take:

- *"C first":* This approach to learning C++ is wasteful of students' time and leads to poor programming practices by forcing students to approach problems with fewer facilities, techniques, and libraries than necessary. C++ provides stronger type checking than C, a standard library with better support for novices, and exceptions for error handling.
- *Bottom-up:* This approach distracts from learning good and effective programming practices. By forcing students to solve problems with insufficient support from the language and libraries, it promotes poor and wasteful programming practices.
- *"If you present something, you must present it fully":* This approach implies a bottom-up approach (by drilling deeper and deeper into every topic touched). It bores novices with technical details they have no interest in and quite likely will not need for years to come. Once you can program, you can look up technical details in a manual. Manuals are good at that, whereas they are awful for initial learning of concepts.
- *Top-down:* This approach, working from first principles toward details, tends to distract readers from the practical aspects of programming and

force them to concentrate on high-level concepts before they have any chance of appreciating their importance. For example, you simply can't appreciate proper software development principles before you have learned how easy it is to make a mistake in a program and how hard it can be to correct it.

- *"Abstract first":* Focusing on general principles and protecting the student from nasty real-world constraints can lead to a disdain for real-world problems, languages, tools, and hardware constraints. Often, this approach is supported by "teaching languages" that cannot be used later and (deliberately) insulate students from hardware and system concerns.
- *"Software engineering principles first":* This approach and the abstract-first approach tend to share the problems of the top-down approach: without concrete examples and practical experience, you simply cannot appreciate the value of abstraction and proper software development practices.
- *"Object-oriented from day one":* Object-oriented programming is one of the best ways of organizing code and programming efforts, but it is not the only effective way. In particular, we feel that a grounding in the basics of types and algorithmic code is a prerequisite for appreciation of the design of classes and class hierarchies. We do use user-defined types (what some people would call "objects") from day one, but we don't show how to design a class until Chapter 6 and don't show a class hierarchy until Chapter 12.
- *"Just believe in magic":* This approach relies on demonstrations of powerful tools and techniques without introducing the novice to the underlying techniques and facilities. This leaves the student guessing – and usually guessing wrong – about why things are the way they are, what it costs to use them, and where they can be reasonably applied. This can lead to overrigid following of familiar patterns of work and become a barrier to further learning.

Naturally, we do not claim that these other approaches are never useful. In fact, we use several of these for specific subtopics where their strengths can be appreciated. However, as general approaches to learning programming aimed at real-world use, we reject them and apply our alternative: concrete-first and depth-first with an emphasis on concepts and techniques.

0.2.2 Programming and programming language

We teach programming first and treat our chosen programming language as secondary, as a tool. Our general approach can be used with any general-purpose programming language. Our primary aim is to help you learn general concepts, principles, and techniques. However, those cannot be appreciated in isolation. For example, details of syntax, the kinds of ideas that can be directly expressed, and tool support differ from programming language to programming language. However, many of the fundamental techniques for producing bug-free code, such as writing logically simple code (Chapters 5 and 6), establishing invariants (§9.4.3), and separating interfaces from implementation details (§9.7 and §14.1–2), vary little from programming language to programming language.

Programming and design techniques must be learned using a programming language. Design, code organization, and debugging are not skills you can acquire in the abstract. You need to write code in some programming language and gain practical experience with that. This implies that you must learn the basics of a programming language. We say "the basics" because the days when you could learn all of a major industrial language in a few weeks are gone for good. The parts of C++ we present were chosen as the subset that most directly supports the production of good code. Also, we present C++ features that you can't avoid encountering either because they are necessary for logical completeness or are common in the C++ community.

0.2.3 Portability

It is common to write C++ to run on a variety of machines. Major C++ applications run on machines we haven't ever heard of! We consider portability and the use of a variety of machine architectures and operating systems most important. Essentially every example in this book is not only ISO Standard C++, but also portable. Unless specifically stated, the code we present should work on every C++ implementation and has been tested on several machines and operating systems.

The details of how to compile, link, and run a C++ program differ from system to system. It would be tedious to mention the details of every system and every compiler each time we need to refer to an implementation issue. In Appendix C, we give the most basic information about getting started using Visual Studio and Microsoft C++ on a Windows machine.

If you have trouble with one of the popular, but rather elaborate, IDEs (integrated development environments), we suggest you try working from the command line; it's surprisingly simple. For example, here is the full set of commands needed to compile, link, and execute a simple program consisting of two source files, **my_file1.cpp** and **my_file2.cpp**, using the GNU C++ compiler on a Unix or Linux system:

```
c++ –o my_program my_file1.cpp my_file2.cpp
./my_program
```

Yes, that really is all it takes.

0.3 Programming and computer science

Is programming all that there is to computer science? Of course not! The only reason we raise this question is that people have been known to be confused about this. We touch upon major topics from computer science, such as algorithms and data structures, but our aim is to teach programming: the design and implementation of programs. That is both more and less than most accepted notions of computer science:

- *More*, because programming involves many technical skills that are not usually considered part of any science
- *Less*, because we do not systematically present the foundation for the parts of computer science we use

The aim of this book is to be part of a course in computer science (if becoming a computer scientist is your aim), to be the foundation for the first of many courses in software construction and maintenance (if your aim is to become a programmer or a software engineer), and in general to be part of a greater whole.

We rely on computer science throughout and we emphasize principles, but we teach programming as a practical skill based on theory and experience, rather than as a science.

0.4 Creativity and problem solving

The primary aim of this book is to help you to express your ideas in code, not to teach you how to get those ideas. Along the way, we give many examples of how we can address a problem, usually through analysis of a problem followed by gradual refinement of a solution. We consider programming itself a form of problem solving: only through complete understanding of a problem and its solution can you express a correct program for it, and only through constructing and testing a program can you be certain that your understanding is complete. Thus, programming is inherently part of an effort to gain understanding. However, we aim to demonstrate this through examples, rather than through "preaching" or presentation of detailed prescriptions for problem solving.

0.5 Request for feedback

We don't think that the perfect textbook can exist; the needs of individuals differ too much for that. However, we'd like to make this book and its supporting materials as good as we can make them. For that, we need feedback; a good textbook cannot be written in isolation from its readers. Please send us reports on errors, typos, unclear text, missing explanations, etc. We'd also appreciate suggestions for better exercises, better examples, and topics to add, topics to delete, etc. Constructive comments will help future readers and we'll post errata on our support website: www.stroustrup.com/Programming.

0.6 References

Along with listing the publications mentioned in this chapter, this section also includes publications you might find helpful.

Becker, Pete, ed. *The C++ Standard*. ISO/IEC 14882:2011.

Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4, Second Edition*. Prentice Hall, 2008. ISBN 0132354160.

Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.

Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley, 2005. ISBN 0321334876.

Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2001. ISBN 0201604647.

Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2002. ISBN 0201795256.

Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.

Stroustrup, Bjarne. "Learning Standard C++ as a New Language." *C/C++ Users Journal,* May 1999.

Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 0321563840.

Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 0321958314.

Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 1999. ISBN 0201615622.

A more comprehensive list of references can be found in the Bibliography section at the back of the book.

0.7 Biographies

You might reasonably ask, "Who are these guys who want to teach me how to program?" So here is some biographical information. I, Bjarne Stroustrup, wrote this book, and together with Lawrence "Pete" Petersen, I designed and taught the university-level beginner's (first-year) course that was developed concurrently with the book, using drafts of the book.

Bjarne Stroustrup

I'm the designer and original implementer of the C++ programming language. I have used the language, and many other programming languages, for a wide variety of programming tasks over the last 40 years or so. I just love elegant and efficient code used in challenging applications, such as robot control, graphics, games, text analysis, and networking. I have taught design, programming, and C++ to people of essentially all abilities and interests. I'm a founding member of the ISO standards committee for C++ where I serve as the chair of the working group for language evolution.

This is my first introductory book. My other books, such as *The C++ Programming Language* and *The Design and Evolution of C++*, were written for experienced programmers.

I was born into a blue-collar (working-class) family in Århus, Denmark, and got my master's degree in mathematics with computer science in my hometown university. My Ph.D. in computer science is from Cambridge University, England. I worked for AT&T for about 25 years, first in the famous Computer Science Research Center of Bell Labs – where Unix, C, C++, and so much more was invented – and later in AT&T Labs–Research.

I'm a member of the U.S. National Academy of Engineering, a Fellow of the ACM, and an IEEE Fellow. As the first computer scientist ever, I received the 2005 William Procter Prize for Scientific Achievement from Sigma Xi (the scientific research society). In 2010, I received the University of Åarhus's oldest and most prestigious honor for contributions to science by a person associated with the university, the *Rigmor og Carl Holst-Knudsens Videnskapspris*. In 2013, I was made Honorary Doctor of Computer Science from the National Research University, ITMO, St. Petersburg, Russia.

I do have a life outside work. I'm married and have two children, one a medical doctor and one a Post-doctoral Research Fellow. I read a lot (including history, science fiction, crime, and current affairs) and like most kinds of music (including classical, rock, blues, and country). Good food with friends is an essential part of life, and I enjoy visiting interesting places and people, all over the world. To be able to enjoy the good food, I run.

For more information, see my home pages: www.stroustrup.com. In particular, there you can find out how to pronounce my name.

Lawrence "Pete" Petersen

In late 2006, Pete introduced himself as follows: "I am a teacher. For almost 20 years, I have taught programming languages at Texas A&M. I have been selected by students for Teaching Excellence Awards five times and in 1996 received the Distinguished Teaching Award from the Alumni Association for the College of Engineering. I am a Fellow of the Wakonse Program for Teaching Excellence and a Fellow of the Academy for Educator Development.

"As the son of an army officer, I was raised on the move. After completing a degree in philosophy at the University of Washington, I served in the army for 22 years as a Field Artillery Officer and as a Research Analyst for Operational Testing. I taught at the Field Artillery Officers' Advanced Course at Fort Sill, Oklahoma, from 1971 to 1973. In 1979 I helped organize a Test Officers' Training Course and taught it as lead instructor at nine different locations across the United States from 1978 to 1981 and from 1985 to 1989.

"In 1991 I formed a small software company that produced management software for university departments until 1999. My interests are in teaching, designing, and programming software that real people can use. I completed master's degrees in industrial engineering at Georgia Tech and in education curriculum and instruction at Texas A&M. I also completed a master's program in microcomputers from NTS. My Ph.D. is in information and operations management from Texas A&M.

"My wife, Barbara, and I live in Bryan, Texas. We like to travel, garden, and entertain; and we spend as much time as we can with our sons and their families, and especially with our grandchildren, Angelina, Carlos, Tess, Avery, Nicholas, and Jordan."

Sadly, Pete died of lung cancer in 2007. Without him, the course would never have succeeded.

Postscript

Most chapters provide a short "postscript" that attempts to give some perspective on the information presented in the chapter. We do that with the realization that the information can be – and often is – daunting and will only be fully comprehended after doing exercises, reading further chapters (which apply the ideas of the chapter), and a later review. *Don't panic!* Relax; this is natural and expected. You won't become an expert in a day, but you can become a reasonably competent programmer as you work your way through the book. On the way, you'll encounter much information, many examples, and many techniques that lots of programmers have found stimulating and fun.

This page intentionally left blank

Vectors and Arrays

> “Caveat emptor!”
>
> **—Good advice**

This chapter describes how vectors are copied and accessed through subscripting. To do that, we discuss copying in general and consider **vector**’s relation to the lower-level notion of arrays. We present arrays’ relation to pointers and consider the problems arising from their use. We also present the five essential operations that must be considered for every type: construction, default construction, copy construction, copy assignment, and destruction. In addition, a container needs a move constructor and a move assignment.

18.1 Introduction

To get into the air, a plane has to accelerate along the runway until it moves fast enough to "jump" into the air. While the plane is lumbering along the runway, it is little more than a particularly heavy and awkward truck. Once in the air, it soars to become an altogether different, elegant, and efficient vehicle. It is in its true element.

In this chapter, we are in the middle of a "run" to gather enough programming language features and techniques to get away from the constraints and difficulties of plain computer memory. We want to get to the point where we can program using types that provide exactly the properties we want based on logical needs. To "get there" we have to overcome a number of fundamental constraints related to access to the bare machine, such as the following:

- An object in memory is of fixed size.
- An object in memory is in one specific place.
- The computer provides only a few fundamental operations on such objects (such as copying a word, adding the values from two words, etc.).

Basically, those are the constraints on the built-in types and operations of C++ (as inherited through C from hardware; see §22.2.5 and Chapter 27). In Chapter 17, we saw the beginnings of a **vector** type that controls all access to its elements and provides us with operations that seem "natural" from the point of view of a user, rather than from the point of view of hardware.

This chapter focuses on the notion of copying. This is an important but rather technical point: What do we mean by copying a nontrivial object? To what extent are the copies independent after a copy operation? What copy operations are there? How do we specify them? And how do they relate to other fundamental operations, such as initialization and cleanup?

Inevitably, we get to discuss how memory is manipulated when we don't have higher-level types such as **vector** and **string**. We examine arrays and pointers, their relationship, their use, and the traps and pitfalls of their use. This is essential information to anyone who gets to work with low-level uses of C++ or C code.

Please note that the details of **vector** are peculiar to **vector**s and the C++ ways of building new higher-level types from lower-level ones. However, every "higher-level" type (**string**, **vector**, **list**, **map**, etc.) in every language is somehow built from the same machine primitives and reflects a variety of resolutions to the fundamental problems described here.

18.2 Initialization

Consider our **vector** as it was at the end of Chapter 17:

```
class vector {
        int sz;                 // the size
        double* elem;           // a pointer to the elements
public:
        vector(int s)                                   // constructor
                :sz{s}, elem{new double[s]} { /* . . . */ }   // allocates memory
        ~vector()                                       // destructor
                { delete[] elem; }                      // deallocates memory
        // . . .
};
```

That's fine, but what if we want to initialize a vector to a set of values that are not defaults? For example:

```
vector v1 = {1.2, 7.89, 12.34 };
```

We can do that, and it is much better than initializing to default values and then assigning the values we really want:

```
vector v2(2);          // tedious and error-prone
v2[0] = 1.2;
v2[1] = 7.89;
v2[2] = 12.34;
```

Compared to **v1**, the "initialization" of **v2** is tedious and error-prone (we deliberately got the number of elements wrong in that code fragment). Using **push_back()** can save us from mentioning the size:

```
vector v3;                  // tedious and repetitive
v2.push_back(1.2);
v2.push_back(7.89);
v2.push_back(12.34);
```

But this is still repetitive, so how do we write a constructor that accepts an initializer list as its argument? A **{ }**-delimited list of elements of type **T** is presented to the programmer as an object of the standard library type **initializer_list<T>**, a list of **T**s, so we can write

```
class vector {
        int sz;                 // the size
        double* elem;           // a pointer to the elements
public:
        vector(int s)           // constructor (s is the element count)
                :sz{s}, elem{new double[sz]}   // uninitialized memory for elements
        {
                for (int i = 0; i<sz; ++i) elem[i] = 0.0;     // initialize
        }

        vector(initializer_list<double> lst)          // initializer-list constructor
                :sz{lst.size()}, elem{new double[sz]}  // uninitialized memory
                                                       // for elements
        {
                copy( lst.begin(),lst.end(),elem); // initialize (using std::copy(); §B.5.2)
        }
        // . . .
};
```

We used the standard library **copy** algorithm (§B.5.2). It copies a sequence of elements specified by its first two arguments (here, the beginning and the end of the **initializer_list**) to a sequence of elements starting with its third argument (here, the **vector**'s elements starting at **elem**).

Now we can write

```
vector v1 = {1,2,3};     // three elements 1.0, 2.0, 3.0
vector v2(3);            // three elements each with the (default) value 0.0
```

Note how we use **( )** for an element count and **{ }** for element lists. We need a notation to distinguish them. For example:

```
vector v1 {3};          // one element with the value 3.0
vector v2(3);           // three elements each with the (default) value 0.0
```

This is not very elegant, but it is effective. If there is a choice, the compiler will interpret a value in a **{ }** list as an element value and pass it to the initializer-list constructor as an element of an **initializer_list**.

In most cases – including all cases we will encounter in this book – the **=** before an **{ }** initializer list is optional, so we can write

```
vector v11 = {1,2,3};   // three elements 1.0, 2.0, 3.0
vector v12 {1,2,3};     // three elements 1.0, 2.0, 3.0
```

The difference is purely one of style.

Note that we pass **initializer_list<double>** by value. That was deliberate and required by the language rules: an **initializer_list** is simply a handle to elements allocated "elsewhere" (see §B.6.4).

18.3 Copying

Consider again our incomplete **vector**:

```
class vector {
        int sz;                 // the size
        double* elem;           // a pointer to the elements
public:
        vector(int s)                                   // constructor
                :sz{s}, elem{new double[s]} { /* . . . */ }    // allocates memory
        ~vector()                                       // destructor
                { delete[] elem; }                      // deallocates memory
        // . . .
};
```

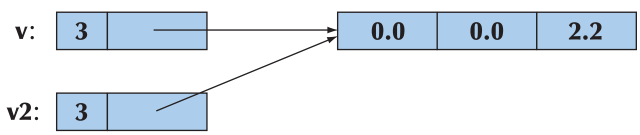
Let's try to copy one of these vectors:

```
void f(int n)
{
        vector v(3);            // define a vector of 3 elements
        v.set(2,2.2);           // set v[2] to 2.2
```

```
        vector v2 = v;          // what happens here?
        // . . .
}
```

Ideally, **v2** becomes a copy of **v** (that is, **=** makes copies); that is, **v2.size()==v.size()** and **v2[i]==v[i]** for all is in the range [**0**:**v.size()**). Furthermore, all memory is returned to the free store upon exit from **f()**. That's what the standard library **vector** does (of course), but it's not what happens for our still-far-too-simple **vector**. Our task is to improve our **vector** to get it to handle such examples correctly, but first let's figure out what our current version actually does. Exactly what does it do wrong? How? And why? Once we know that, we can probably fix the problems. More importantly, we have a chance to recognize and avoid similar problems when we see them in other contexts.

The default meaning of copying for a class is "Copy all the data members." That often makes perfect sense. For example, we copy a **Point** by copying its coordinates. But for a pointer member, just copying the members causes problems. In particular, for the **vector**s in our example, it means that after the copy, we have **v.sz==v2.sz** and **v.elem==v2.elem** so that our **vector**s look like this:



That is, **v2** doesn't have a copy of **v**'s elements; it shares **v**'s elements. We could write

```
v.set(1,99);            // set v[1] to 99
v2.set(0,88);           // set v2[0] to 88
cout << v.get(0) << ' ' << v2.get(1);
```

The result would be the output **88 99**. That wasn't what we wanted. Had there been no "hidden" connection between **v** and **v2**, we would have gotten the output **0 0**, because we never wrote to **v[0]** or to **v2[1]**. You could argue that the behavior we got is "interesting," "neat!" or "sometimes useful," but that is not what we intended or what the standard library **vector** provides. Also, what happens when we return from **f()** is an unmitigated disaster. Then, the destructors for **v** and **v2** are implicitly called; **v**'s destructor frees the storage used for the elements using

```
delete[] elem;
```

and so does **v2**'s destructor. Since **elem** points to the same memory location in both **v** and **v2**, that memory will be freed twice with likely disastrous results (§17.4.6).

18.3.1 Copy constructors

So, what do we do? We'll do the obvious: provide a copy operation that copies the elements and make sure that this copy operation gets called when we initialize one **vector** with another.

Initialization of objects of a class is done by a constructor. So, we need a constructor that copies. Unsurprisingly, such a constructor is called a *copy constructor*. It is defined to take as its argument a reference to the object from which to copy. So, for class **vector** we need

```
vector(const vector&);
```

This constructor will be called when we try to initialize one **vector** with another. We pass by reference because we (obviously) don't want to copy the argument of the constructor that defines copying. We pass by **const** reference because we don't want to modify our argument (§8.5.6). So we refine **vector** like this:

```
class vector {
        int sz;
        double* elem;
public:
        vector(const vector&) ;            // copy constructor: define copy
        // . . .
};
```

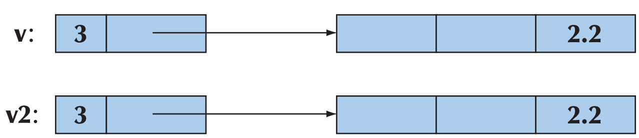
The copy constructor sets the number of elements (**sz**) and allocates memory for the elements (initializing **elem**) before copying element values from the argument **vector**:

```
vector:: vector(const vector& arg)
// allocate elements, then initialize them by copying
        :sz{arg.sz}, elem{new double[arg.sz]}
{
        copy(arg,arg+sz,elem);        // std::copy(); see §B.5.2
}
```

Given this copy constructor, consider again our example:

```
vector v2 = v;
```

This definition will initialize **v2** by a call of **vector**'s copy constructor with **v** as its argument. Again given a **vector** with three elements, we now get



Given that, the destructor can do the right thing. Each set of elements is correctly freed. Obviously, the two **vector**s are now independent so that we can change the value of elements in **v** without affecting **v2** and vice versa. For example:

```
v.set(1,99);              // set v[1] to 99
v2.set(0,88);             // set v2[0] to 88
cout << v.get(0) << ' ' << v2.get(1);
```

This will output **0 0**.

Instead of saying

```
vector v2 = v;
```

we could equally well have said

```
vector v2 {v};
```

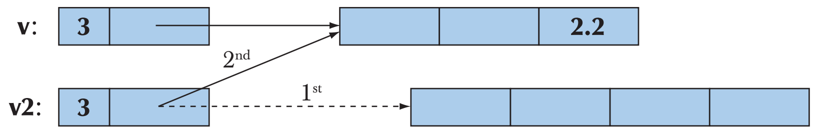
When **v** (the initializer) and **v2** (the variable being initialized) are of the same type and that type has copying conventionally defined, those two notations mean exactly the same thing and you can use whichever notation you like better.

18.3.2 Copy assignments

We handle copy construction (initialization), but we can also copy **vector**s by assignment. As with copy initialization, the default meaning of copy assignment is memberwise copy, so with **vector** as defined so far, assignment will cause a double deletion (exactly as shown for copy constructors in §18.3.1) plus a memory leak. For example:

```
void f2(int n)
{
        vector v(3);          // define a vector
        v.set(2,2.2);
        vector v2(4);
        v2 = v;               // assignment: what happens here?
        // . . .
}
```

We would like **v2** to be a copy of **v** (and that's what the standard library **vector** does), but since we have said nothing about the meaning of assignment of our **vector**, the default assignment is used; that is, the assignment is a memberwise copy so that **v2**'s **sz** and **elem** become identical to **v**'s **sz** and **elem**, respectively. We can illustrate that like this:



When we leave **f2()**, we have the same disaster as we had when leaving **f()** in §18.3 before we added the copy constructor: the elements pointed to by both **v** and **v2** are freed twice (using **delete[ ]**). In addition, we have leaked the memory initially allocated for **v2**'s four elements. We "forgot" to free those. The remedy for this copy assignment is fundamentally the same as for the copy initialization (§18.3.1). We define an assignment that copies properly:

```
class vector {
        int sz;
        double* elem;
public:
        vector& operator=(const vector&) ;       // copy assignment
        // . . .
};

vector& vector::operator=(const vector& a)
        // make this vector a copy of a
{
        double* p = new double[a.sz];            // allocate new space
        copy(a.elem,a.elem+a.sz,elem);           // copy elements
        delete[] elem;                           // deallocate old space
        elem = p;                                // now we can reset elem
        sz = a.sz;
        return *this;                   // return a self-reference (see §17.10)
}
```

Assignment is a bit more complicated than construction because we must deal with the old elements. Our basic strategy is to make a copy of the elements from the source **vector**:

```
        double* p = new double[a.sz];            // allocate new space
        copy(a.elem,a.elem+a.sz,elem);           // copy elements
```

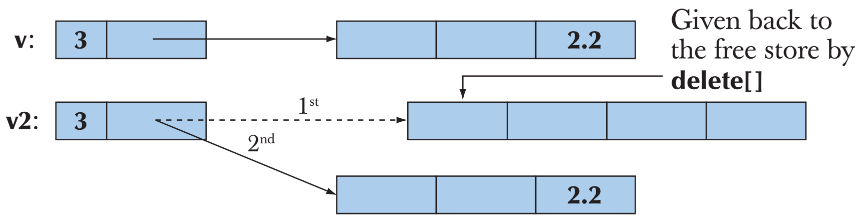
Then we free the old elements from the target **vector**:

```
delete[] elem;                                   // deallocate old space
```

Finally, we let **elem** point to the new elements:

```
elem = p;                    // now we can reset elem
sz = a.sz;
```

We can represent the result graphically like this:



We now have a **vector** that doesn't leak memory and doesn't free (**delete[]**) any memory twice.

When implementing the assignment, you could consider simplifying the code by freeing the memory for the old elements before creating the copy, but it is usually a very good idea not to throw away information before you know that you can replace it. Also, if you did that, strange things would happen if you assigned a **vector** to itself:

```
vector v(10);
        v = v;    // self-assignment
```

Please check that our implementation handles that case correctly (if not with optimal efficiency).

18.3.3 Copy terminology

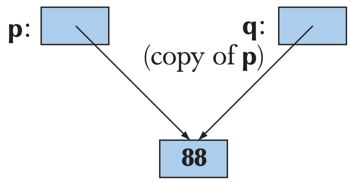
Copying is an issue in most programs and in most programming languages. The basic issue is whether you copy a pointer (or reference) or copy the information pointed to (referred to):

- *Shallow copy* copies only a pointer so that the two pointers now refer to the same object. That's what pointers and references do.
- *Deep copy* copies what a pointer points to so that the two pointers now refer to distinct objects. That's what **vector**s, **string**s, etc. do. We define copy constructors and copy assignments when we want deep copy for objects of our classes.

Here is an example of shallow copy:

```
int* p = new int{77};
int* q = p;             // copy the pointer p
*p = 88;                // change the value of the int pointed to by p and q
```

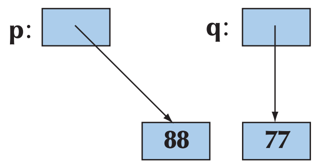
We can illustrate that like this:



In contrast, we can do a deep copy:

```
int* p = new int{77};
int* q = new int{*p};   // allocate a new int, then copy the value pointed to by p
*p = 88;                // change the value of the int pointed to by p
```

We can illustrate that like this:



Using this terminology, we can say that the problem with our original **vector** was that it did a shallow copy, rather than copying the elements pointed to by its **elem** pointer. Our improved **vector**, like the standard library **vector**, does a deep copy by allocating new space for the elements and copying their values. Types that provide shallow copy (like pointers and references) are said to have *pointer semantics* or *reference semantics* (they copy addresses). Types that provide deep copy (like **string** and **vector**) are said to have *value semantics* (they copy the values pointed to). From a user perspective, types with value semantics behave as if no pointers were involved – just values that can be copied. One way of thinking of types with value semantics is that they "work just like integers" as far as copying is concerned.

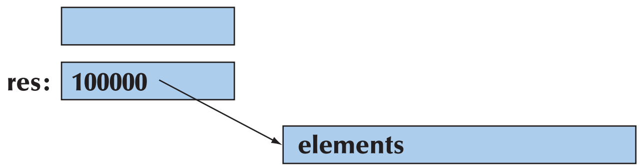
18.3.4 Moving

If a **vector** has a lot of elements, it can be expensive to copy. So, we should copy **vector**s only when we need to. Consider an example:

```
vector fill(istream& is)
{
```

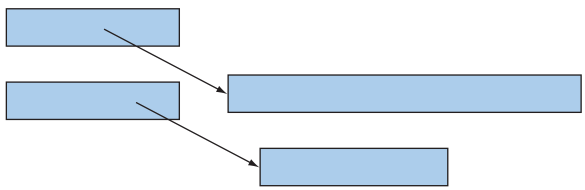
```
        vector res;
        for (double x; is>>x; ) res.push_back(x);
        return res;
}

void use()
{
        vector vec = fill(cin);
        // ... use vec ...
}
```

Here, we fill the local vector **res** from the input stream and return it to **use()**. Copying **res** out of **fill()** and into **vec** could be expensive. But why copy? We don't want a copy! We can never use the original (**res**) after the return. In fact, **res** is destroyed as part of the return from **fill()**. So how can we avoid the copy? Consider again how a vector is represented in memory:



We would like to "steal" the representation of **res** to use for **vec**. In other words, we would like **vec** to refer to the elements of **res** without any copy.

After moving **res**'s element pointer and element count to **vec**, **res** holds no elements. We have successfully moved the value from **res** out of **fill()** to **vec**. Now, **res** can be destroyed (simply and efficiently) without any undesirable side effects:



We have successfully moved 100,000 **double**s out of **fill()** and into its caller at the cost of four single-word assignments.

How do we express such a move in C++ code? We define move operations to complement the copy operations:

```
class vector {
        int sz;
        double* elem;
```

```
public:
        vector(vector&& a);              // move constructor
        vector& operator=(vector&&);     // move assignment
        // . . .
        };
```

The funny **&&** notation is called an "rvalue reference." We use it for defining move operations. Note that move operations do not take **const** arguments; that is, we write **(vector&&)** and not **(const vector&&)**. Part of the purpose of a move operation is to modify the source, to make it "empty." The definitions of move operations tend to be simple. They tend to be simpler and more efficient than their copy equivalents. For **vector**, we get

```
vector::vector(vector&& a)
        :sz{a.sz}, elem{a.elem}          // copy a's elem and sz
{
        a.sz = 0;                        // make a the empty vector
        a.elem = nullptr;
}

vector& vector::operator=(vector&& a) // move a to this vector
{
        delete[] elem;                   // deallocate old space
        elem = a.elem;                   // copy a's elem and sz
        sz = a.sz;
        a.elem = nullptr;                // make a the empty vector
        a.sz = 0;
        return *this;                    // return a self-reference (see §17.10)
}
```

By defining a move constructor, we make it easy and cheap to move around large amounts of information, such as a vector with many elements. Consider again:

```
vector fill(istream& is)
{
        vector res;
        for (double x; is>>x; ) res.push_back(x);
        return res;
}
```

The move constructor is implicitly used to implement the return. The compiler knows that the local value returned (**res**) is about to go out of scope, so it can move from it, rather than copying.

The importance of move constructors is that we do not have to deal with pointers or references to get large amounts of information out of a function. Consider this flawed (but conventional) alternative:

```
vector* fill2(istream& is)
{
	vector* res = new vector;
	for (double x; is>>x; ) res->push_back(x);
	return res;
}

void use2()
{
	vector* vec = fill(cin);
	// ... use vec ...
	delete vec;
}
```

Now we have to remember to delete the **vector**. As described in §17.4.6, deleting objects placed on the free store is not as easy to do consistently and correctly as it might seem.

18.4 Essential operations

We have now reached the point where we can discuss how to decide which constructors a class should have, whether it should have a destructor, and whether you need to provide copy and move operations. There are seven essential operations to consider:

- Constructors from one or more arguments
- Default constructor
- Copy constructor (copy object of same type)
- Copy assignment (copy object of same type)
- Move constructor (move object of same type)
- Move assignment (move object of same type)
- Destructor

Usually we need one or more constructors that take arguments needed to initialize an object. For example:

```
string s {"cat.jpg"};                     // initialize s to the character string "cat.jpg"
Image ii {Point{200,300},"cat.jpg"};  // initialize a Point with the
                                          // coordinates{200,300},
                                          // then display the contents of file
                                          // cat.jpg at that Point
```

The meaning/use of an initializer is completely up to the constructor. The standard **string**'s constructor uses a character string as an initial value, whereas **Image**'s constructor uses the string as the name of a file to open. Usually we use a constructor to establish an invariant (§9.4.3). If we can't define a good invariant for a class that its constructors can establish, we probably have a poorly designed class or a plain data structure.

Constructors that take arguments are as varied as the classes they serve. The remaining operations have more regular patterns.

How do we know if a class needs a default constructor? We need a default constructor if we want to be able to make objects of the class without specifying an initializer. The most common example is when we want to put objects of a class into a standard library **vector**. The following works only because we have default values for **int**, **string**, and **vector<int>**:

```
vector<double> vi(10);          // vector of 10 doubles, each initialized to 0.0
vector<string> vs(10);          // vector of 10 strings, each initialized to ""
vector<vector<int>> vvi(10);    // vector of 10 vectors, each initialized to vector{}
```

So, having a default constructor is often useful. The question then becomes: "When does it make sense to have a default constructor?" An answer is: "When we can establish the invariant for the class with a meaningful and obvious default value." For value types, such as **int** and **double**, the obvious value is **0** (for **double**, that becomes **0.0**). For **string**, the empty **string**, **""**, is the obvious choice. For **vector**, the empty **vector** serves well. For every type **T**, **T{}** is the default value, if a default exists. For example, **double{}** is **0.0**, **string{}** is **""**, and **vector<int>{}** is the empty **vector** of **int**s.

A class needs a destructor if it acquires resources. A resource is something you "get from somewhere" and that you must give back once you have finished using it. The obvious example is memory that you get from the free store (using **new**) and have to give back to the free store (using **delete** or **delete[]**). Our **vector** acquires memory to hold its elements, so it has to give that memory back; therefore, it needs a destructor. Other resources that you might encounter as your programs increase in ambition and sophistication are files (if you open one, you also have to close it), locks, thread handles, and sockets (for communication with processes and remote computers).

Another sign that a class needs a destructor is simply that it has members that are pointers or references. If a class has a pointer or a reference member, it often needs a destructor and copy operations.

A class that needs a destructor almost always also needs a copy constructor and a copy assignment. The reason is simply that if an object has acquired a resource (and has a pointer member pointing to it), the default meaning of copy (shallow, memberwise copy) is almost certainly wrong. Again, **vector** is the classic example.

Similarly, a class that needs a destructor almost always also needs a move constructor and a move assignment. The reason is simply that if an object has acquired a resource (and has a pointer member pointing to it), the default meaning of copy (shallow, memberwise copy) is almost certainly wrong and the usual remedy (copy operations that duplicate the complete object state) can be expensive. Again, **vector** is the classic example.

In addition, a base class for which a derived class may have a destructor needs a **virtual** destructor (§17.5.2).

18.4.1 Explicit constructors

A constructor that takes a single argument defines a conversion from its argument type to its class. This can be most useful. For example:

```
class complex {
public:
        complex(double);            // defines double-to-complex conversion
        complex(double,double);
        // . . .
};

complex z1 = 3.14;                  // OK: convert 3.14 to (3.14,0)
complex z2 = complex{1.2, 3.4};
```

However, implicit conversions should be used sparingly and with caution, because they can cause unexpected and undesirable effects. For example, our **vector**, as defined so far, has a constructor that takes an **int**. This implies that it defines a conversion from **int** to **vector**. For example:

```
class vector {
        // . . .
        vector(int);
        // . . .
};
```

```
vector v = 10;          // odd: makes a vector of 10 doubles
v = 20;                 // eh? Assigns a new vector of 20 doubles to v

void f(const vector&);
f(10);                  // eh? Calls f with a new vector of 10 doubles
```

It seems we are getting more than we have bargained for. Fortunately, it is simple to suppress this use of a constructor as an implicit conversion. A constructor-defined **explicit** provides only the usual construction semantics and not the implicit conversions. For example:

```
class vector {
        // . . .
        explicit vector(int);
        // . . .
};

vector v = 10;          // error: no int-to-vector conversion
v = 20;                 // error: no int-to-vector conversion
vector v0(10);          // OK

void f(const vector&);
f(10);                  // error: no int-to-vector<double> conversion
f(vector(10));          // OK
```

To avoid surprising conversions, we – and the standard – define **vector**'s single-argument constructors to be **explicit**. It's a pity that constructors are not **explicit** by default; if in doubt, make any constructor that can be invoked with a single argument **explicit**.

18.4.2 Debugging constructors and destructors

Constructors and destructors are invoked at well-defined and predictable points of a program's execution. However, we don't always write **explicit** calls, such as **vector(2)**; rather we do something, such as declaring a **vector**, passing a **vector** as a by-value argument, or creating a **vector** on the free store using **new**. This can cause confusion for people who think in terms of syntax. There is not just a single syntax that triggers a constructor. It is simpler to think of constructors and destructors this way:

- Whenever an object of type **X** is created, one of **X**'s constructors is invoked.
- Whenever an object of type **X** is destroyed, **X**'s destructor is invoked.

A destructor is called whenever an object of its class is destroyed; that happens when names go out of scope, the program terminates, or **delete** is used on a pointer to an object. A constructor (some appropriate constructor) is invoked whenever an object of its class is created; that happens when a variable is initialized, when an object is created using **new** (except for built-in types), and whenever an object is copied.

But when does that happen? A good way to get a feel for that is to add print statements to constructors, assignment operations, and destructors and then just try. For example:

```
struct X {              // simple test class
    int val;

        void out(const string& s, int nv)
                { cerr << this << "–>" << s << ": " << val << " (" << nv << ")\n"; }

        X(){ out("X()",0); val=0; }                          // default constructor
        X(int v) { val=v; out( "X(int)",v); }
        X(const X& x){ val=x.val; out("X(X&) ",x.val); }     // copy constructor
        X& operator=(const X& a)                             // copy assignment
                { out("X::operator=()",a.val); val=a.val; return *this; }
        ~X() { out("~X()",0); }                              // destructor
};
```

Anything we do with this **X** will leave a trace that we can study. For example:

```
X glob(2);              // a global variable

X copy(X a) { return a; }

X copy2(X a) { X aa = a; return aa; }

X& ref_to(X& a) { return a; }

X* make(int i) { X a(i); return new X(a); }

struct XX { X a; X b; };

int main()
{
        X loc {4};      // local variable
        X loc2 {loc};   // copy construction
```

```
        loc = X{5};               // copy assignment
        loc2 = copy(loc);         // call by value and return
        loc2 = copy2(loc);
        X loc3 {6};
        X& r = ref_to(loc);       // call by reference and return
        delete make(7);
        delete make(8);
        vector<X> v(4);           // default values
        XX loc4;
        X* p = new X{9};          // an X on the free store
        delete p;
        X* pp = new X[5];         // an array of Xs on the free store
        delete[] pp;
}
```

Try executing that.

TRY THIS

We really mean it: do run this example and make sure you understand the result. If you do, you'll understand most of what there is to know about construction and destruction of objects.

Depending on the quality of your compiler, you may note some "missing copies" relating to our calls of **copy()** and **copy2()**. We (humans) can see that those functions do nothing: they just copy a value unmodified from input to output. If a compiler is smart enough to notice that, it is allowed to eliminate the calls to the copy constructor. In other words, a compiler is allowed to assume that a copy constructor copies and does nothing but copy. Some compilers are smart enough to eliminate many spurious copies. However, compilers are not guaranteed to be that smart, so if you want portable performance, consider move operations (§18.3.4).

Now consider: Why should we bother with this "silly class **X**"? It's a bit like the finger exercises that musicians have to do. After doing them, other things – things that matter – become easier. Also, if you have problems with constructors and destructors, you can insert such print statements in constructors for your real classes to see that they work as intended. For larger programs, this exact kind of tracing becomes tedious, but similar techniques apply. For example, you can determine whether you have a memory leak by seeing if the number of constructions minus the number of destructions equals zero. Forgetting to define copy constructors and copy assignments for classes that allocate memory or hold pointers to objects is a common – and easily avoidable – source of problems.

If your problems get too big to handle by such simple means, you will have learned enough to be able to start using the professional tools for finding such problems; they are often referred to as "leak detectors." The ideal, of course, is not to leak memory by using techniques that avoid such leaks.

18.5 Access to vector elements

So far (§17.6), we have used **set()** and **get()** member functions to access elements. Such uses are verbose and ugly. We want our usual subscript notation: **v[i]**. The way to get that is to define a member function called **operator[ ]**. Here is our first (naive) try:

```
class vector {
        int sz;                 // the size
        double* elem;           // a pointer to the elements
public:
        // . . .
        double operator[ ](int n) { return elem[n]; }      // return element
};
```

That looks good and especially it looks simple, but unfortunately it is too simple. Letting the subscript operator (**operator[ ]()**) return a value enables reading but not writing of elements:

```
vector v(10);
double x = v[2];          // fine
v[3] = x;                 // error: v[3] is not an lvalue
```

Here, **v[i]** is interpreted as a call **v.operator[ ](i)**, and that call returns the value of **v**'s element number **i**. For this overly naive **vector**, **v[3]** is a floating-point value, not a floating-point variable.

TRY THIS

Make a version of this **vector** that is complete enough to compile and see what error message your compiler produces for **v[3]=x;**.

Our next try is to let **operator[ ]** return a pointer to the appropriate element:

```
class vector {
        int sz;                 // the size
        double* elem;           // a pointer to the elements
```

```
public:
        // . . .
        double* operator[](int n) { return &elem[n]; }      // return pointer
};
```

Given that definition, we can write

```
vector v(10);
for (int i=0; i<v.size(); ++i) {          // works, but still too ugly
        *v[i] = i;
        cout << *v[i];
}
```

Here, **v[i]** is interpreted as a call **v.operator[](i)**, and that call returns a pointer to **v**'s element number **i**. The problem is that we have to write ***** to dereference that pointer to get to the element. That's almost as bad as having to write **set()** and **get()**. Returning a reference from the subscript operator solves this problem:

```
class vector {
        // . . .
        double& operator[ ](int n) { return elem[n]; }     // return reference
};
```

Now we can write

```
vector v(10);
for (int i=0; i<v.size(); ++i) {          // works!
        v[i] = i;                         // v[i] returns a reference element i
        cout << v[i];
}
```

We have achieved the conventional notation: **v[i]** is interpreted as a call **v.operator[](i)**, and that returns a reference to **v**'s element number **i**.

18.5.1 Overloading on const

The **operator[]()** defined so far has a problem: it cannot be invoked for a **const vector**. For example:

```
void f(const vector& cv)
{
        double d = cv[1];                 // error, but should be fine
        cv[1] = 2.0;                      // error (as it should be)
}
```

The reason is that our **vector::operator[]()** could potentially change a **vector**. It doesn't, but the compiler doesn't know that because we "forgot" to tell it. The solution is to provide a version that is a **const** member function (see §9.7.4). That's easily done:

```
class vector {
    // . . .
    double& operator[](int n);        // for non-const vectors
    double operator[](int n) const;   // for const vectors
};
```

We obviously couldn't return a **double&** from the **const** version, so we returned a **double** value. We could equally well have returned a **const double&**, but since a **double** is a small object there would be no point in returning a reference (§8.5.6), so we decided to pass it back by value. We can now write

```
void ff(const vector& cv, vector& v)
{
    double d = cv[1];      // fine (uses the const [])
    cv[1] = 2.0;           // error (uses the const [])
    double d = v[1];       // fine (uses the non-const [])
    v[1] = 2.0;            // fine (uses the non-const [])
}
```

Since **vector**s are often passed by **const** reference, this **const** version of **operator[]()** is an essential addition.

18.6 Arrays

For a while, we have used *array* to refer to a sequence of objects allocated on the free store. We can also allocate arrays elsewhere as named variables. In fact, they are common

- As global variables (but global variables are most often a bad idea)
- As local variables (but arrays have serious limitations there)
- As function arguments (but an array doesn't know its own size)
- As class members (but member arrays can be hard to initialize)

Now, you might have detected that we have a not-so-subtle bias in favor of **vector**s over arrays. Use **std::vector** where you have a choice – and you have a choice in most contexts. However, arrays existed long before **vector**s and are roughly equivalent to what is offered in other languages (notably C), so you must know arrays, and know them well, to be able to cope with older code and with code written by people who don't appreciate the advantages of **vector**.

So, what is an array? How do we define an array? How do we use an array? An *array* is a homogeneous sequence of objects allocated in contiguous memory; that is, all elements of an array have the same type and there are no gaps between the objects of the sequence. The elements of an array are numbered from 0 upward. In a declaration, an array is indicated by "square brackets":

```
const int max = 100;
int gai[max];           // a global array (of 100 ints); "lives forever"

void f(int n)
{
      char lac[20];     // local array; "lives" until the end of scope
      int lai[60];
      double lad[n];    // error: array size not a constant
      // . . .
}
```

Note the limitation: the number of elements of a named array must be known at compile time. If you want the number of elements to be a variable, you must put it on the free store and access it through a pointer. That's what **vector** does with its array of elements.

Just like the arrays on the free store, we access named arrays using the subscript and dereference operators (**[ ]** and *****). For example:

```
void f2()
{
      char lac[20];     // local array; "lives" until the end of scope

      lac[7] = 'a';
      *lac = 'b';       // equivalent to lac[0]='b'

      lac[–2] = 'b';    // huh?
      lac[200] = 'c';   // huh?
}
```

This function compiles, but we know that "compiles" doesn't mean "works correctly." The use of **[ ]** is obvious, but there is no range checking, so **f2()** compiles, and the result of writing to **lac[–2]** and **lac[200]** is (as for all out-of-range access) usually disastrous. Don't do it. Arrays do not range check. Again, we are dealing directly with physical memory here; don't expect "system support."

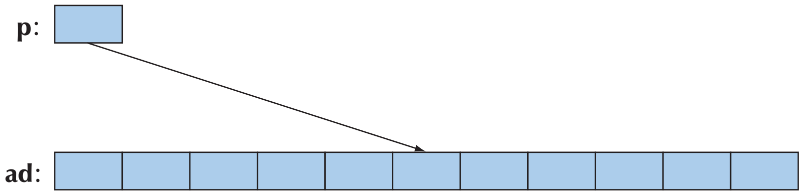
But couldn't the compiler see that **lac** has just 20 elements so that **lac[200]** is an error? A compiler could, but as far as we know no production compiler does. The problem is that keeping track of array bounds at compile time is impossible in general, and catching errors in the simplest cases (like the one above) only is not very helpful.

18.6.1 Pointers to array elements

A pointer can point to an element of an array. Consider:

```
double ad[10];
double* p = &ad[5];          // point to ad[5]
```

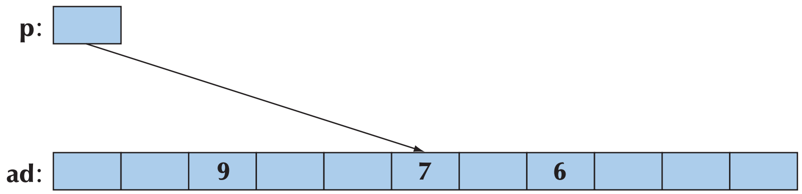
We now have a pointer **p** to the **double** known as **ad[5]**:



We can subscript and dereference that pointer:

```
*p =7;
p[2] = 6;
p[–3] = 9;
```

We get

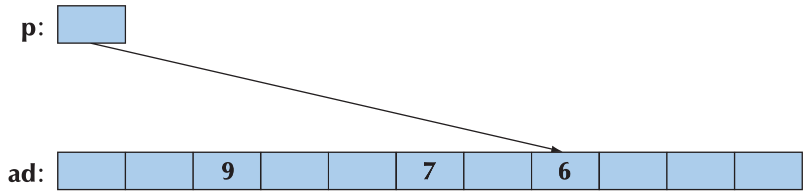


That is, we can subscript the pointer with both positive and negative numbers. As long as the resulting element is in range, all is well. However, access outside the range of the array pointed into is illegal (as with free-store-allocated arrays; see §17.4.3). Typically, access outside an array is not detected by the compiler and (sooner or later) is disastrous.

Once a pointer points into an array, addition and subscripting can be used to make it point to another element of the array. For example:

```
p += 2;          // move p 2 elements to the right
```

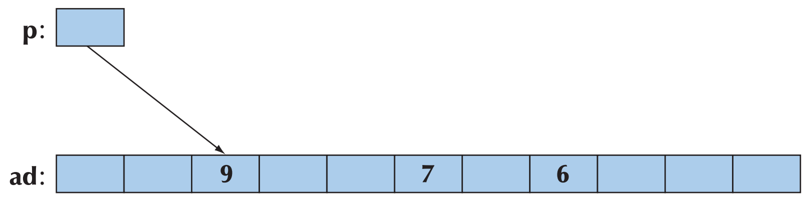
We get



And

```
p -= 5;          // move p 5 elements to the left
```

We get



Using **+**, **–**, **+=**, and **–=** to move pointers around is called *pointer arithmetic*. Obviously, if we do that, we have to take great care to ensure that the result is not a pointer to memory outside the array:

```
p += 1000;           // insane: p points into an array with just 10 elements
double d = *p;       // illegal: probably a bad value
                     // (definitely an unpredictable value)
*p = 12.34;          // illegal: probably scrambles some unknown data
```

Unfortunately, not all bad bugs involving pointer arithmetic are that easy to spot. The best policy is usually simply to avoid pointer arithmetic.

The most common use of pointer arithmetic is incrementing a pointer (using **++**) to point to the next element and decrementing a pointer (using **––**) to point to the previous element. For example, we could print the value of **ad**'s elements like this:

```
for (double* p = &ad[0]; p<&ad[10]; ++p) cout << *p << '\n';
```

Or backward:

```
for (double* p = &ad[9]; p>=&ad[0]; --p) cout << *p << '\n';
```

This use of pointer arithmetic is not uncommon. However, we find the last ("backward") example quite easy to get wrong. Why **&ad[9]** and not **&ad[10]**? Why **>=** and not **>**? These examples could equally well (and equally efficiently) be done using subscripting. Such examples could be done equally well using subscripting into a **vector**, which is more easily range checked.

Note that most real-world uses of pointer arithmetic involve a pointer passed as a function argument. In that case, the compiler doesn't have a clue how many elements are in the array pointed into: you are on your own. That is a situation we prefer to stay away from whenever we can.

Why does C++ have (allow) pointer arithmetic at all? It can be such a bother and doesn't provide anything new once we have subscripting. For example:

```
double* p1 = &ad[0];
double* p2 = p1+7;
double* p3 = &p1[7];
if (p2 != p3) cout << "impossible!\n";
```

Mainly, the reason is historical. These rules were crafted for C decades ago and can't be removed without breaking a lot of code. Partly, there can be some convenience gained by using pointer arithmetic in some important low-level applications, such as memory managers.

18.6.2 Pointers and arrays

The name of an array refers to all the elements of the array. Consider:

```
char ch[100];
```

The size of **ch**, **sizeof(ch)**, is 100. However, the name of an array turns into ("decays to") a pointer with the slightest excuse. For example:

```
char* p = ch;
```

Here **p** is initialized to **&ch[0]** and **sizeof(p)** is something like 4 (not 100).

This can be useful. For example, consider a function **strlen()** that counts the number of characters in a zero-terminated array of characters:

```
int strlen(const char* p)       // similar to the standard library strlen()
{
        int count = 0;
        while (*p) { ++count; ++p; }
        return count;
}
```

We can now call this with **strlen(ch)** as well as **strlen(&ch[0])**. You might point out that this is a very minor notational advantage, and we'd have to agree.

One reason for having array names convert to pointers is to avoid accidentally passing large amounts of data by value. Consider:

```
int strlen(const char a[])      // similar to the standard library strlen()
{
        int count = 0;
        while (a[count]) { ++count; }
        return count;
}

char lots [100000];

void f()
{
        int nchar = strlen(lots);
        // . . .
}
```

Naively (and quite reasonably), you might expect this call to copy the 100,000 characters specified as the argument to **strlen()**, but that's not what happens. Instead, the argument declaration **char p[]** is considered equivalent to **char* p**, and the call **strlen(lots)** is considered equivalent to **strlen(&lots[0])**. This saves you from an expensive copy operation, but it should surprise you. Why should it surprise you? Because in every other case, when you pass an object and don't explicitly declare an argument to be passed by reference (§8.5.3–6), that object is copied.

Note that the pointer you get from treating the name of an array as a pointer to its first element is a value and not a variable, so you cannot assign to it:

```
char ac[10];
ac = new char [20];             // error: no assignment to array name
&ac[0] = new char [20];         // error: no assignment to pointer value
```

Finally! A problem that the compiler will catch!

As a consequence of this implicit array-name-to-pointer conversion, you can't even copy arrays using assignment:

```
int x[100];
int y[100];
// . . .
x = y;                          // error
int z[100] = y;                 // error
```

This is consistent, but often a bother. If you need to copy an array, you must write some more elaborate code to do so. For example:

```
for (int i=0; i<100; ++i) x[i]=y[i];     // copy 100 ints
memcpy(x,y,100*sizeof(int));             // copy 100*sizeof(int) bytes
copy(y,y+100, x);                        // copy 100 ints
```

Note that the C language doesn't support anything like **vector**, so in C, you must use arrays extensively. This implies that a lot of C++ code uses arrays (§27.1.2). In particular, C-style strings (zero-terminated arrays of characters; see §27.5) are very common.

If we want assignment, we have to use something like the standard library **vector**. The **vector** equivalent to the copying code above is

```
vector<int> x(100);
vector<int> y(100);
// . . .
x = y;                          // copy 100 ints
```

18.6.3 Array initialization

An array of **char**s can be initialized with a string literal. For example:

```
char ac[] = "Beorn";            // array of 6 chars
```

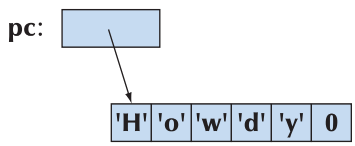
Count those characters. There are five, but **ac** becomes an array of six characters because the compiler adds a terminating zero character at the end of a string literal:

ac:	'B'	'e'	'o'	'r'	'n'	0

A zero-terminated string is the norm in C and many systems. We call such a zero-terminated array of characters a *C-style string*. All string literals are C-style strings. For example:

```
char* pc = "Howdy";                 // pc points to an array of 6 chars
```

Graphically:



Note that the **char** with the numeric value **0** is not the character **'0'** or any other letter or digit. The purpose of that terminating zero is to allow functions to find the end of the string. Remember: An array does not know its size. Relying on the terminating zero convention, we can write

```
int strlen(const char* p)           // similar to the standard library strlen()
{
        int n = 0;
        while (p[n]) ++n;
        return n;
}
```

Actually, we don't have to define **strlen()** because it is a standard library function defined in the **<string.h>** header (§27.5, §B.11.3). Note that **strlen()** counts the characters, but not the terminating **0**; that is, you need *n*+1 **char**s to store *n* characters in a C-style string.

Only character arrays can be initialized by literal strings, but all arrays can be initialized by a list of values of their element type. For example:

```
int ai[] = { 1, 2, 3, 4, 5, 6 };         // array of 6 ints
int ai2[100] = {0,1,2,3,4,5,6,7,8,9};    // the last 90 elements are initialized to 0
double ad[100] = { };                    // all elements initialized to 0.0
char chars[] = {'a', 'b', 'c'};          // no terminating 0!
```

Note that the number of elements of **ai** is six (not seven) and the number of elements for **chars** is three (not four) – the "add a **0** at the end" rule is for literal character strings only. If an array isn't given a size, that size is deduced from the initializer list. That's a rather useful feature. If there are fewer initializer values than array elements (as in the definitions of **ai2** and **ad**), the remaining elements are initialized by the element type's default value.

18.6.4 Pointer problems

Like arrays, pointers are often overused and misused. Often, the problems people get themselves into involve both pointers and arrays, so we'll summarize the problems here. In particular, all serious problems with pointers involve trying to access something that isn't an object of the expected type, and many of those problems involve access outside the bounds of an array. Here we will consider

- Access through the null pointer
- Access through an uninitialized pointer
- Access off the end of an array
- Access to a deallocated object
- Access to an object that has gone out of scope

In all cases, the practical problem for the programmer is that the actual access looks perfectly innocent; it is "just" that the pointer hasn't been given a value that makes the use valid. Worse (in the case of a write through the pointer), the problem may manifest itself only a long time later when some apparently unrelated object has been corrupted. Let's consider examples:

Don't access through the null pointer:

```
int* p = nullptr;
*p = 7;            // ouch!
```

Obviously, in real-world programs, this typically occurs when there is some code in between the initialization and the use. In particular, passing **p** to a function and receiving it as the result from a function are common examples. We prefer not to pass null pointers around, but if you have to, test for the null pointer before use:

```
int* p = fct_that_can_return_a_nullptr();

if (p == nullptr) {
        // do something
}
else {
        // use p
        *p = 7;
}
```

and

```
void fct_that_can_receive_a_nullptr(int* p)
{
        if (p == nullptr) {
                // do something
        }
        else {
                // use p
                *p = 7;
        }
}
```

Using references (§17.9.1) and using exceptions to signal errors (§5.6 and §19.5) are the main tools for avoiding null pointers.

Do initialize your pointers:

```
int* p;
*p = 9;          // ouch!
```

In particular, don't forget to initialize pointers that are class members.

Don't access nonexistent array elements:

```
int a[10];
int* p = &a[10];
*p = 11;         // ouch!
a[10] = 12;      // ouch!
```

Be careful with the first and last elements of a loop, and try not to pass arrays around as pointers to their first elements. Instead use **vector**s. If you really must use an array in more than one function (passing it as an argument), then be extra careful and pass its size along.

Don't access through a deleted pointer:

```
int* p = new int{7};
// . . .
delete p;
// . . .
*p = 13;         // ouch!
```

The **delete p** or the code after it may have scribbled all over ***p** or used it for something else. Of all of these problems, we consider this one the hardest to systematically avoid. The most effective defense against this problem is not to have "naked" **new**s that require "naked" **delete**s: use **new** and **delete** in constructors and destructors or use a container, such as **Vector_ref** (§E.4), to handle **delete**s.

Don't return a pointer to a local variable:

```
int* f()
{
        int x = 7;
        // . . .
        return &x;
}

// . . .

int* p = f();
// . . .
*p = 15;                // ouch!
```

The return from **f()** or the code after it may have scribbled all over ***p** or used it for something else. The reason for that is that the local variables of a function are allocated (on the stack) upon entry to the function and deallocated again at the exit from the function. In particular, destructors are called for local variables of classes with destructors (§17.5.1). Compilers could catch most problems related to returning pointers to local variables, but few do.

Consider a logically equivalent example:

```
vector& ff()
{
        vector x(7);    // 7 elements
        // . . .
        return x;
}   // the vector x is destroyed here

// . . .

vector& p = ff();
// . . .
p[4] = 15;              // ouch!
```

Quite a few compilers catch this variant of the return problem.

It is common for programmers to underestimate these problems. However, many experienced programmers have been defeated by the innumerable variations and combinations of these simple array and pointer problems. The solution is not to litter your code with pointers, arrays, **new**s, and **delete**s. If you do, "being careful" simply isn't enough in realistically sized programs. Instead, rely on vectors, RAII ("Resource Acquisition Is Initialization"; see §19.5), and other systematic approaches to the management of memory and other resources.

18.7 Examples: palindrome

Enough technical examples! Let's try a little puzzle. A *palindrome* is a word that is spelled the same from both ends. For example, *anna*, *petep*, and *malayalam* are palindromes, whereas *ida* and *homesick* are not. There are two basic ways of determining whether a word is a palindrome:

- Make a copy of the letters in reverse order and compare that copy to the original.
- See if the first letter is the same as the last, then see if the second letter is the same as the second to last, and keep going until you reach the middle.

Here, we'll take the second approach. There are many ways of expressing this idea in code depending on how we represent the word and how we keep track of how far we have come with the comparison of characters. We'll write a little program that tests whether words are palindromes in a few different ways just to see how different language features affect the way the code looks and works.

18.7.1 Palindromes using string

First, we try a version using the standard library **string** with **int** indices to keep track of how far we have come with our comparison:

```
bool is_palindrome(const string& s)
{
	int first = 0;              // index of first letter
	int last = s.length()–1;    // index of last letter
	while (first < last) {      // we haven't reached the middle
		if (s[first]!=s[last]) return false;
		++first;            // move forward
		––last;             // move backward
	}
	return true;
}
```

We return **true** if we reach the middle without finding a difference. We suggest that you look at this code to convince yourself that it is correct when there are no letters in the string, just one letter in the string, an even number of letters in the string, and an odd number of letters in the string. Of course, we should not just rely on logic to see that our code is correct. We should also test. We can exercise **is_palindrome()** like this:

```
int main()
{
        for (string s; cin>>s; ) {
                cout << s << " is";
                if (!is_palindrome(s)) cout << " not";
                cout << " a palindrome\n";
        }
}
```

Basically, the reason we are using a **string** is that "**string**s are good for dealing with words." It is simple to read a whitespace-separated word into a string, and a **string** knows its size. Had we wanted to test **is_palindrome()** with strings containing whitespace, we could have read using **getline()** (§11.5). That would have shown *ah ha* and *as df fd sa* to be palindromes.

18.7.2 Palindromes using arrays

What if we didn't have **string**s (or **vector**s), so that we had to use an array to store the characters? Let's see:

```
bool is_palindrome(const char s[], int n)
        // s points to the first character of an array of n characters
{
        int first = 0;                  // index of first letter
        int last = n–1;                 // index of last letter
        while (first < last) {          // we haven't reached the middle
                if (s[first]!=s[last]) return false;
                ++first;                // move forward
                ––last;                 // move backward
        }
        return true;
}
```

To exercise **is_palindrome()**, we first have to get characters read into the array. One way to do that safely (i.e., without risk of overflowing the array) is like this:

```
istream& read_word(istream& is, char* buffer, int max)
        // read at most max–1 characters from is into buffer
```

```
{
        is.width(max);          // read at most max–1 characters in the next >>
        is >> buffer;           // read whitespace-terminated word,
                                // add zero after the last character read into buffer
        return is;
}
```

Setting the **istream**'s width appropriately prevents buffer overflow for the next **>>** operation. Unfortunately, it also means that we don't know if the read terminated by whitespace or by the buffer being full (so that we need to read more characters). Also, who remembers the details of the behavior of **width()** for input? The standard library **string** and **vector** are really better as input buffers because they expand to fit the amount of input. The terminating **0** character is needed because most popular operations on arrays of characters (C-style strings) assume 0 termination. Using **read_word()** we can write

```
int main()
{
        constexpr int max = 128;
        for (char s[max]; read_word(cin,s,max); ) {
                cout << s << " is";
                if (!is_palindrome(s,strlen(s))) cout << " not";
                cout << " a palindrome\n";
        }
}
```

The **strlen(s)** call returns the number of characters in the array after the call of **read_word()**, and **cout<<s** outputs the characters in the array up to the terminating **0**.

We consider this "array solution" significantly messier than the "**string** solution," and it gets much worse if we try to seriously deal with the possibility of long strings. See exercise 10.

18.7.3 Palindromes using pointers

Instead of using indices to identify characters, we could use pointers:

```
bool is_palindrome(const char* first, const char* last)
        // first points to the first letter, last to the last letter
{
        while (first < last) {          // we haven't reached the middle
                if (*first!=*last) return false;
                ++first;                // move forward
                ––last;                 // move backward
```

```
        }
        return true;
}
```

Note that we can actually increment and decrement pointers. Increment makes a pointer point to the next element of an array and decrement makes a pointer point to the previous element. If the array doesn't have such a next element or previous element, you have a serious uncaught out-of-range error. That's another problem with pointers.

We call this is_palindrome() like this:

```
int main()
{
        const int max = 128;
        for (char s[max]; read_word(cin,s,max); ) {
                cout << s << " is";
                if (!is_palindrome(&s[0],&s[strlen(s)–1])) cout << " not";
                cout << " a palindrome\n";
        }
}
```

Just for fun, we rewrite is_palindrome() like this:

```
bool is_palindrome(const char* first, const char* last)
        // first points to the first letter, last to the last letter
{
        if (first<last) {
                if (*first!=*last) return false;
                return is_palindrome(first+1,last–1);
        }
        return true;
}
```

This code becomes obvious when we rephrase the definition of *palindrome*: a word is a palindrome if the first and the last characters are the same and if the substring you get by removing the first and the last characters is a palindrome.

Drill

In this chapter, we have two drills: one to exercise arrays and one to exercise **vector**s in roughly the same manner. Do both and compare the effort involved in each.

Array drill:

1. Define a global **int** array **ga** of ten **int**s initialized to 1, 2, 4, 8, 16, etc.
2. Define a function **f()** taking an **int** array argument and an **int** argument indicating the number of elements in the array.
3. In **f()**:
   a. Define a local **int** array **la** of ten **int**s.
   b. Copy the values from **ga** into **la**.
   c. Print out the elements of **la**.
   d. Define a pointer **p** to **int** and initialize it with an array allocated on the free store with the same number of elements as the argument array.
   e. Copy the values from the argument array into the free-store array.
   f. Print out the elements of the free-store array.
   g. Deallocate the free-store array.
4. In **main()**:
   a. Call **f()** with **ga** as its argument.
   b. Define an array **aa** with ten elements, and initialize it with the first ten factorial values (1, 2*1, 3*2*1, 4*3*2*1, etc.).
   c. Call **f()** with **aa** as its argument.

Standard library vector drill:

1. Define a global **vector<int> gv**; initialize it with ten **int**s, 1, 2, 4, 8, 16, etc.
2. Define a function **f()** taking a **vector<int>** argument.
3. In **f()**:
   a. Define a local **vector<int> lv** with the same number of elements as the argument **vector**.
   b. Copy the values from **gv** into **lv**.
   c. Print out the elements of **lv**.
   d. Define a local **vector<int> lv2**; initialize it to be a copy of the argument **vector**.
   e. Print out the elements of **lv2**.
4. In **main()**:
   a. Call **f()** with **gv** as its argument.
   b. Define a **vector<int> vv**, and initialize it with the first ten factorial values (1, 2*1, 3*2*1, 4*3*2*1, etc.).
   c. Call **f()** with **vv** as its argument.

Review

1. What does "Caveat emptor!" mean?
2. What is the default meaning of copying for class objects?
3. When is the default meaning of copying of class objects appropriate? When is it inappropriate?
4. What is a copy constructor?
5. What is a copy assignment?
6. What is the difference between copy assignment and copy initialization?
7. What is shallow copy? What is deep copy?
8. How does the copy of a **vector** compare to its source?
9. What are the five "essential operations" for a class?
10. What is an **explicit** constructor? Where would you prefer one over the (default) alternative?
11. What operations may be invoked implicitly for a class object?
12. What is an array?
13. How do you copy an array?
14. How do you initialize an array?
15. When should you prefer a pointer argument over a reference argument? Why?
16. What is a C-style string?
17. What is a palindrome?

Terms

array	deep copy	move assignment
array initialization	default constructor	move construction
copy assignment	essential operations	palindrome
copy constructor	**explicit** constructor	shallow copy

Exercises

1. Write a function, **char* strdup(const char*)**, that copies a C-style string into memory it allocates on the free store. Do not use any standard library functions. Do not use subscripting; use the dereference operator ***** instead.
2. Write a function, **char* findx(const char* s, const char* x)**, that finds the first occurrence of the C-style string **x** in **s**. Do not use any standard library functions. Do not use subscripting; use the dereference operator ***** instead.
3. Write a function, **int strcmp(const char* s1, const char* s2)**, that compares C-style strings. Let it return a negative number if **s1** is lexicographically

before **s2**, zero if **s1** equals **s2**, and a positive number if **s1** is lexicographically after **s2**. Do not use any standard library functions. Do not use subscripting; use the dereference operator * instead.

4. Consider what happens if you give **strdup()**, **findx()**, and **strcmp()** an argument that is not a C-style string. Try it! First figure out how to get a **char*** that doesn't point to a zero-terminated array of characters and then use it (never do this in real – non-experimental – code; it can create havoc). Try it with free-store-allocated and stack-allocated "fake C-style strings." If the results still look reasonable, turn off debug mode. Redesign and re-implement those three functions so that they take another argument giving the maximum number of elements allowed in argument strings. Then, test that with correct C-style strings and "bad" strings.
5. Write a function, **string cat_dot(const string& s1, const string& s2)**, that concatenates two strings with a dot in between. For example, **cat_dot("Niels", "Bohr")** will return a string containing **Niels.Bohr**.
6. Modify **cat_dot()** from the previous exercise to take a string to be used as the separator (rather than dot) as its third argument.
7. Write versions of the **cat_dot()**s from the previous exercises to take C-style strings as arguments and return a free-store-allocated C-style string as the result. Do not use standard library functions or types in the implementation. Test these functions with several strings. Be sure to free (using **delete**) all the memory you allocated from free store (using **new**). Compare the effort involved in this exercise with the effort involved for exercises 5 and 6.
8. Rewrite all the functions in §18.7 to use the approach of making a backward copy of the string and then comparing; for example, take **"home"**, generate **"emoh"**, and compare those two strings to see that they are different, so *home* isn't a palindrome.
9. Consider the memory layout in §17.4. Write a program that tells the order in which static storage, the stack, and the free store are laid out in memory. In which direction does the stack grow: upward toward higher addresses or downward toward lower addresses? In an array on the free store, are elements with higher indices allocated at higher or lower addresses?
10. Look at the "array solution" to the palindrome problem in §18.7.2. Fix it to deal with long strings by (a) reporting if an input string was too long and (b) allowing an arbitrarily long string. Comment on the complexity of the two versions.
11. Look up (e.g., on the web) *skip list* and implement that kind of list. This is not an easy exercise.
12. Implement a version of the game "Hunt the Wumpus." "Hunt the Wumpus" (or just "Wump") is a simple (non-graphical) computer game originally invented by Gregory Yob. The basic premise is that a rather smelly

monster lives in a dark cave consisting of connected rooms. Your job is to slay the wumpus using bow and arrow. In addition to the wumpus, the cave has two hazards: bottomless pits and giant bats. If you enter a room with a bottomless pit, it's the end of the game for you. If you enter a room with a bat, the bat picks you up and drops you into another room. If you enter the room with the wumpus or he enters yours, he eats you. When you enter a room you will be told if a hazard is nearby:

"I smell the wumpus": It's in an adjoining room.

"I feel a breeze": One of the adjoining rooms is a bottomless pit.

"I hear a bat": A giant bat is in an adjoining room.

For your convenience, rooms are numbered. Every room is connected by tunnels to three other rooms. When entering a room, you are told something like "You are in room 12; there are tunnels to rooms 1, 13, and 4; move or shoot?" Possible answers are **m13** ("Move to room 13") and **s13–4–3** ("Shoot an arrow through rooms 13, 4, and 3"). The range of an arrow is three rooms. At the start of the game, you have five arrows. The snag about shooting is that it wakes up the wumpus and he moves to a room adjoining the one he was in – that could be your room.

Probably the trickiest part of the exercise is to make the cave by selecting which rooms are connected with which other rooms. You'll probably want to use a random number generator (e.g., **randint()** from **std_lib_facilities.h**) to make different runs of the program use different caves and to move around the bats and the wumpus. Hint: Be sure to have a way to produce a debug output of the state of the cave.

Postscript

The standard library **vector** is built from lower-level memory management facilities, such as pointers and arrays, and its primary role is to help us avoid the complexities of those facilities. Whenever we design a class, we must consider initialization, copying, and destruction.

This page intentionally left blank

Index

C

D

E

M

N

P

S

T

U

V

W

X

Z