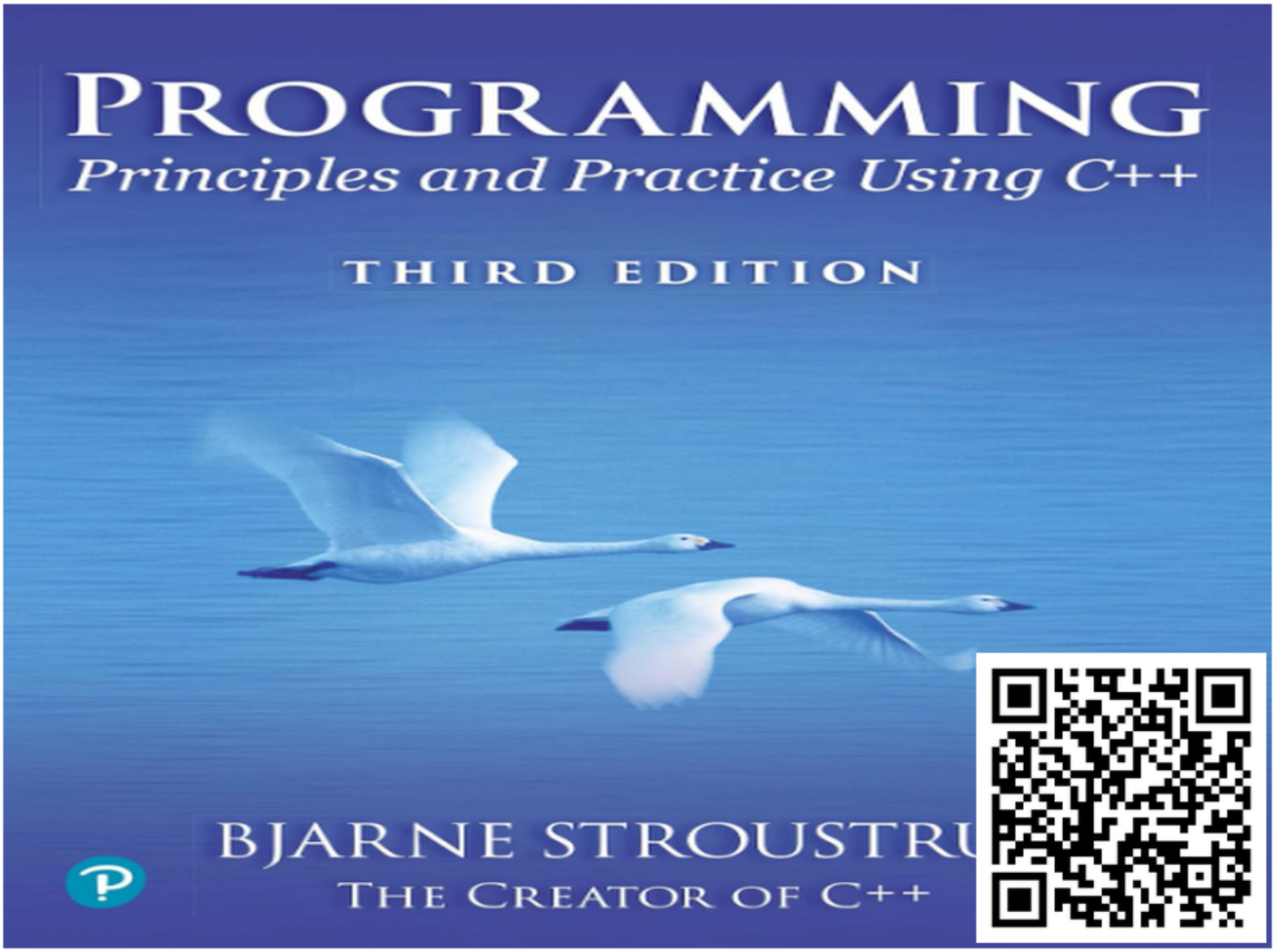

PROGRAMMING

Principles and Practice Using C++

THIRD EDITION

BJARNE STROUSTRUP

THE CREATOR OF C++

Programming: Principles and Practice Using C++ Third Edition

Bjarne Stroustrup

Addison-Wesley

Hoboken, New Jersey

Cover photo by Photowood Inc./Corbis.
Author photo courtesy of Bjarne Stroustrup.
Page 294: ‘‘Promenade a Skagen’’ by Peder Severin Kroyer.
Page 308: Photo of NASA’s Ingenuity Mars Helicopter, The National Aeronautics and Space Administration (NASA).
Page 354: Photo of Hurricane Rita as seen from space, The National Oceanic and Atmospheric Administration (NOAA).

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2024932369

This book was typeset in Times and Helvetica by the author.

ISBN-13: 978-0-13-830868-1
ISBN-10: 0-13-83086-3
First printing, May 2024
$PrintCode

Contents

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. It can also be used by someone with some programming knowledge who wants a more thorough grounding in programming principles and contemporary C++.

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.

There are many kinds of programming. This book aims to serve those who want to write non-trivial 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, and provide exercises for it. If you just want to understand toy programs or write programs that just call code provided by others, you can get along with far less than I present. In such cases, you will

probably also be better served by a language that's simpler than C++. 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.

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 become a good programmer 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!

There is more to programming – much more – than following a few rules and reading the manual. This book is not focused on "the syntax of C++." C++ is used to illustrate fundamental concepts. Understanding the fundamental ideals, principles, and techniques is the essence of a good programmer. Also, "the fundamentals" are what last: they will still be essential long after today's programming languages and tools have evolved or been replaced.

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

Previous Editions

The third edition of *Programming: Principles and Practice Using C++* is about half the size of the second edition. Students having to carry the book will appreciate the lighter weight. The reason for the reduced size is simply that more information about C++ and its standard library is available on the Web. The essence of the book that is generally used in a course in programming is in this third edition ("PPP3"), updated to C++20 plus a bit of C++23. The fourth part of the previous edition ("PPP2") was designed to provide extra information for students to look up when needed and is available on the Web:

- Chapter 1: Computers, People, and Programming
- Chapter 11: Customizing Input and Output
- Chapter 22: Ideas and History
- Chapter 23 Text Manipulation
- Chapter 24: Numerics
- Chapter 25: Embedded Systems Programming
- Chapter 26: Testing
- Chapter 27: The C Programming Language
- Glossary

Where I felt it useful to reference these chapters, the references look like this: PPP2.Ch22 or PPP2.§27.1.

Acknowledgments

Special thanks to the people who reviewed drafts of this book and suggested many improvements: Clovis L. Tondo, Jose Daniel Garcia Sanchez, J.C. van Winkel, and Ville Voutilainen. Also, Ville Voutilainen did the non-trivial mapping of the GUI/Graphics interface library to Qt, making it portable to an amazing range of systems.

Also, thanks to the many people who contributed to the first and second editions of this book. Many of their comments are reflected in this third edition.

This page intentionally left blank

0

Notes to the Reader

$e^{i\pi} + 1$
– Leonhard Euler

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. Before writing any code, read "PPP support" (§0.4). A teacher will find most parts immediately useful. If you are reading this book as a novice, please don't try to understand everything. 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 three parts:

- Part I (Chapter 1 to Chapter 8) 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 (Chapter 9 to Chapter 14) first 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, we show 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). As part of that, we introduce the fundamental principles and techniques of object-oriented programming.
- Part III (Chapter 15 to Chapter 21) focuses on the C++ standard library's containers and algorithms framework (often referred to as the STL). We show how containers (such as **vector**, **list**, and **map**) are implemented and used. In doing so, we introduce low-level facilities such as pointers, arrays, and dynamic memory. We also show how to handle errors using exceptions and how to parameterize our classes and functions using templates. As part of that, we introduce the fundamental principles and techniques of generic programming. We also demonstrate the design and use of standard-library algorithms (such as **sort**, **find**, and **inner_product**).

The order of topics is determined by programming techniques, rather than programming language features.

CC 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:

- **CC**: concepts and techniques (this paragraph is an example of that)
- **AA**: advice
- **XX**: warning

The use of **CC**, **AA**, and **XX**, rather than a single token in different colors, is to help where colors are not easy to distinguish.

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 mean "you, the author, and teachers," working together through a problem, as we might have done had we been in the same room. I use "I" when I refer to my own work or personal opinions.

AA 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. 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.

A 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, a language 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. CC

0.1.2 Drills, exercises, etc.

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

- *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.

- *Try this*: 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 but focused narrowly on the topic that precedes it. If you pass a *Try this* without trying it out – 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.

In addition, at the end of each chapter we offer some help to solidify what's learned:

- *Review*: 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.
- *Terms*: A 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 term means.
- *Postscript*: A paragraph intended to provide some perspective for the material presented.

In addition, we recommend that you 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 (e.g., while progressing through the later chapters of the book). Most people find such projects the most fun and that they tie everything together.

CC 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?

AA 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 become an expert at programming in four months than you should expect to become 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 project developing code to be used by someone else; preferably guided by an experienced developer. After that, or (even better) in parallel with a project, read either a professional-level general textbook, a more specialized book relating to the needs of your project, or a textbook focusing on a particular aspect of C++ (such as algorithms, graphics, scientific computation, finance, or games); see §0.6.

AA 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. Why? No large program is written in a single language. Also,

different languages typically differ in the way code is thought about and programs are constructed. Design techniques, availability of libraries, and the way programs are built differ, sometimes dramatically. Even when the syntaxes of two languages are similar, the similarity is typically only skin deep. Performance, detection of errors, and constraints on what can be expressed typically differ. This is similar to the ways natural languages and cultures differ. Knowing only a single language and a single culture implies the danger of thinking that "the way we do things" is the only way or the only good way. That way opportunities are missed, and sub-optimal programs are produced. One of the best ways to avoid such problems is to know several languages (programming languages and natural languages).

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)
- Files and stream input and output (I/O)
- Memory management
- Design and programming ideals
- The C++ standard library
- Software development strategies

To keep the book lighter than the small laptop on which it is written, some supplementary topics from the second edition are placed on the Web (§0.4.1):

- Computers, People, and Programming (PPP2.Ch1)
- Ideals and History (PPP2.Ch22)
- Text manipulation (incl. Regular expression matching) (PPP2.Ch23)
- Numerics (PPP2.Ch24)
- Embedded systems programming (PPP2.Ch25)
- C-language programming techniques (PPP2.Ch27)

Working our way through the chapters, 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.

CC We characterize our approach as "depth-first." It is also "concrete-first" and "concept-based." First, we quickly (well, relatively quickly, Chapter 1 to Chapter 9) 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! 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.

AA 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!

XX 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 learn a foreign language. We encourage you to seek help from teachers, friends, colleagues, 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 your initial skills to broaden your base of knowledge. We use examples and exercises to solidify your understanding, and to provide a conceptual base for programming.

AA 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 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 other sources, mostly on the Web (§0.4.1). 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. Remember, though, to consider every Web resource highly suspect until you have reason to believe better of it. Many an authoritative-looking Web site 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 Web site: **www.stroustrup.com/programming.html**.

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.

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 specifically language-technical is taken from a real-world use. Our basic tone is that of professionals addressing (future) professionals.

C++ rests on two pillars:

- *Efficient direct access to machine resources*: making C++ effective for low-level, machine-near, programming as is essential in many application domains.
- *Powerful (Zero-overhead) abstraction mechanisms*: making it possible to escape the error-prone low-level programming by providing elegant, flexible, and type-and-resource-safe, yet efficient facilities needed for higher-level programming.

This book teaches both levels. We use the implementation of higher-level abstractions as our primary examples to introduce low-level language features and programming techniques. The aim is always to write code at the highest level affordable, but that often requires a foundation built using lower-level facilities and techniques. We aim for you to master both levels.

0.2.1 A note to students

Many thousands of first-year university students taught using the first two editions of this book had never before seen a line of code in their lives. Most succeeded, so you can do it, too. **AA**

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 must practice to master.

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 chapter, look at the "Computers, People, and Programming" and "Ideals and History" chapters posted on the Web (§0.4.1). 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.

Please don't be too impatient. Learning any major new and valuable skill takes time.

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.

0.2.2 A note to teachers

CC

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, 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, we expect it to be taught alongside other courses as part of a well-rounded introduction.

Many students like to get an idea why subjects are taught and why they are taught in the way they are. Please try to convey my teaching philosophy, general approach, etc. to your students along the way. Also, to motivate students, please present short examples of areas and applications where C++ is used extensively, such as aerospace, medicine, games, animation, cars, finance, and scientific computation.

0.3 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. The code for this edition of the book uses contemporary C++, C++20 (plus a bit of C++23). If your compiler does not support C++20 [C++20], 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.

On the other hand, you may be in an environment where you are able to use only C++14 or C++17. Most of the contents of this book will still apply, but you'll have trouble with features introduced in C++20:

- modules (§7.7.1). Instead of modules use header files (§7.7.2). In particular, use `#include "PPPheaders.h"` to compile our examples and your exercises, rather than `#include "PPP.h"` (§0.4).
- ranges (§20.7). Use explicit iterators, rather than ranges. For example, `sort(v.begin(),v.end())` rather than `ranges::sort(v)`. If/when that gets tedious, write your own ranges versions of your favorite algorithms (§21.1).
- `span` (§16.4.1). Fall back on the old "pointer and size" technique. For example, `void f(int* p, int n);` rather than `void f(span<int> s);` and do your own range checking as needed.
- concepts (§18.1.3). Use plain `template<typename T>` and hope for the best. The error messages from that for simple mistakes can be horrendous.

0.3.1 Portability

CC

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 the use of C++ on 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. By *portable*, we mean that we make no assumptions about the computer, the operating system, and the compiler beyond that an up-to-date standard-conforming C++ implementation is available. 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. Also, most systems offer you a choice of compilers and tools. Explaining the many and often mutating tool sets is beyond the scope of the book. We might add some such information to the PPP support Web site (§0.4).

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 Linux system:

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

Yes, that really is all it takes.

Another way to get started is to use a build system, such as Cmake (§0.4). However, that path is best taken when there are someone experienced who can guide those first steps.

0.3.2 Guarantees

Except when illustrating errors, the code in this book is type-safe (an object is used only according to its definition). We follow the rules of *The C++ Core Guidelines* to simplify programming and eliminate common errors. You can find the Core Guidelines on the Web [CG] and rule checkers are available when you need guaranteed conformance.

We don't recommend that you delve into this while still a novice, but consider it reassuring that the recommended styles and techniques illustrated in this book have industrial backing. Once you are comfortable with C++ and understand the potential errors (say after Chapter 16), we suggest you read the introduction to the CG and try one of the CG checkers to see how they can eliminate errors before they make it into running code.

0.3.3 A brief history of C++

I started the design and implementation of C++ in late 1979 and supported my first user about six months later. The initial features included classes with constructors and destructors (§8.4.2, §15.5), and function-argument declarations (§3.5.2). Initially, the language was called *C with Classes*, but to avoid confusion with C, it was renamed *C++* in 1984.

The basic idea of C++ was to combine C's ability to utilize hardware efficiently (e.g., device drivers, memory managers, and process schedulers) [K&R] with Simula's facilities for organizing code (notably classes and derived classes) [Simula]. I needed that for a project where I wanted to build a distributed Unix. Had I succeeded, it might have become the first Unix cluster, but the development of C++ "distracted" me from that.

In 1985, the first implementation of a C++ compiler and foundation library was shipped commercially. I wrote most of that and most of its documentation. The first book on C++, *The C++ Programming Language* [TC++PL], was published simultaneously. Then, the language supported what was called data abstraction and object-oriented programming (§12.3, §12.5). In addition, it had feeble support for generic programming (§21.1.2).

In the late 1980s, I worked on the design of exceptions (§4.6) and templates (Chapter 18). The templates were aimed to support *generic programming* along the lines of the work of Alex Stepanov [AS,2009].

In 1989, several large corporations decided that we needed an ISO standard for C++. Together with Margaret Ellis, I wrote the book that became the base document for C++'s standardization "The ARM" [ARM]. The first ISO standard was approved by 20 nations in 1998 and is known as C++98. For a decade, C++98 supported a massive growth in C++ use and gave much valuable feedback to its further evolution. In addition to the language, the standard specifies an extensive standard library. In C++98 the most significant standard-library component was the STL providing iterators (§19.3.2), containers (such as **vector** (§3.6) and **map** (§20.2)), and algorithms (§21).

C++11 was a significant upgrade that added improved facilities for compile-time computation (§3.3.1), lambdas (§13.3.3, §21.2.3), and formalized support for concurrency. Concurrency had been used in C++ from the earliest days, but that interesting and important topic is beyond the scope of this book. Eventually, see [AW,2019]. The C++11 standard library added many useful components, notably random number generation (§4.7.5) and resource-management pointers (unique_ptr (§18.5.2) and shared_ptr; §18.5.3)).

C++14 and C++17 added many useful features without adding support for significantly new programming styles.

C++20 [C++20] was a major improvement of C++, about as significant as C++11 and coming close to meeting my ideals for C++ as articulated in *The Design and Evolution of C++* in 1994 [DnE]. Among many extensions, it added modules (§7.7.1), concepts (§18.1.3), coroutines (beyond the scope of this book), and ranges (§20.7).

These changes over decades have been evolutionary with a great concern for backwards compatibility. I have small programs from the 1980s that still run today. Where old code fails to compile or work correctly, the reason is usually changes to the operating systems or third-party libraries. This gives a degree of stability that is considered a major feature by organizations that maintain software that is in use for decades.

For a more thorough discussion of the design and evolution of C++, see *The Design and Evolution of C++* [DnE] and my three *History of Programming* papers [HOPL-2] [HOPL-3] [HOPL-4]. Those were not written for novices, though.

0.4 PPP support

All the code in this book is ISO standard C++. To start compiling and running the examples, add two lines at the start of the code:

```
import std;
using namespace std;
```

This makes the standard library available.

Unfortunately, the standard does not guarantee range checking for containers, such as the standard **vector**, and most implementations do not enforce it by default. Typically, enforcement must be enabled by options that differ between different compilers. We consider range checking essential to simplify learning and minimize frustration. So, we supply a module **PPP_support** that makes a version of the C++ standard library with guaranteed range checking for subscripting available (see **www.stroustrup.com/programming.html**). So instead of directly using module **std** directly, use:

```
#include "PPP.h"
```

We also supply **"PPPheaders.h"** as a similar version to **"PPP.h"** for people who don't have access to a compiler with good module support. This supplies less of the C++ standard library than **"PPP.h"** and will compile slower.

In addition to the range checking, **PPP_support** provides a convenient **error()** function and a simplified interface to the standard random number facilities that many students have found useful in the past. We strongly recommend using **PPP.h** consistently.

Some people have commented about our use of a support header for PPP1 and PPP2 that "using a non-standard header is not real C++." Well, it is because the content of those headers is 100% ISO C++ and doesn't change the meaning of correct programs. We consider it important that our PPP support does a decent job at helping you to avoid non-portable code and surprising behavior. Also, writing libraries that makes it easier to support good and efficient code is one of the main uses of C++. **PPP_support** is just one simple example of that.

AA If you cannot download the files supporting PPP, or have trouble getting them to compile, use the standard library directly, but try to figure out how to enable range checking. All major C++ implementations have an option for that, but it is not always easy to find and enable it. For all startup problems, it is best to take advice from someone experienced.

In addition, when you get to Chapter 10 and need to run Graphics and GUI code, you need to install the Qt graphics/GUI system and an interface library specifically designed for this book. See _display.system_ and **www.stroustrup.com/programming.html**.

0.4.1 Web resources

There is an overwhelming amount of material about C++, both text and videos, on the Web. Unfortunately, it is of varying quality, much is aimed at advanced users, and much is outdated. So use it with care and a healthy dose of skepticism.

AA The support site for this book is **www.stroustrup.com/programming.html**. There, you can find

- The **PPP_support** module source code (§0.4).
- The **PPP.h** and **PPPheaders.h** headers (§0.4).
- Some installation guidance for PPP support.
- Some code examples.
- Errata.
- Chapters from PPP2 (the second edition of *Programming: Principles and Practice using C++*) [PPP2] that were eliminated from the print version to save weight and because alternative sources have become available. These chapters are available at **www.stroustrup.com/programming.html** and referred to in the PPP3 text like this: PPP2.Ch22 or PPP2.§22.1.2.

Other Web resources:

- My Web site **www.stroustrup.com** contains a lot of material related to C++.
- The C++ Foundation's Web site **www.isocpp.org** has various useful and interesting information, much about the standardization but also a stream of articles and news items.
- I recommend **cppreference.com** as an on-line reference. I use it myself daily to look up obscure details of the language and the standard library. I don't recommend using it as a tutorial.
- The major C++ implementers, such as Clang, GCC, and Microsoft, offer free downloads of good versions of their products (**www.stroustrup.com/compilers.html**). All have options enforcing range checking of subscripting.
- There are several Web sites offering (free) on-line C++ compilation, e.g., the *compiler explorer* **https://godbolt.org**. These are easy to use and very useful for testing out small examples and for seeing how different compilers and different versions of compilers handle source code.
- For guidance on how to use contemporary C++, see *The C++ Core Guidelines*: The C++ Core Guidelines (**https://github.com/isocpp/CppCoreGuidelines**) [CG] and its small support library (**https://github.com/microsoft/GSL**). Except when illustrating mistakes, the CG is used in this book.
- For Chapter 10 to Chapter 14, we use Qt as the basis of our graphics and GUI code: **www.qt.io**.

0.5 Author biography

You might reasonably ask: "Who are you to think you can help me to learn how to program?" Here is a canned bio:

> Bjarne Stroustrup is the designer and original implementer of C++ as well as the author of *The C++ Programming Language (4th edition)*, *A Tour of C++ (3rd edition)*, *Programming: Principles and Practice Using C++ (3rd edition)*, and many popular and academic publications. He is a professor of Computer Science at Columbia University in New York City. Dr. Stroustrup is a member of the US National Academy of Engineering, and an IEEE, ACM, and CHM fellow. He received the 2018 Charles Stark Draper Prize, the IEEE Computer Society's 2018 Computer Pioneer Award, and the 2017 IET Faraday Medal. Before joining Columbia University, he was a University Distinguished Professor at Texas A&M University and a Technical Fellow and Managing Director at Morgan Stanley. He did much of his most important work in Bell Labs. His research interests include distributed systems, design, programming techniques, software development tools, and programming languages. To make C++ a stable and up-to-date base for real-world software development, he has been a leading figure with the ISO C++ standards effort for more than 30 years. He holds a master's in mathematics from Aarhus University, where he is an honorary professor in the Computer Science Department, and a PhD in Computer Science from Cambridge University, where he is an honorary fellow of Churchill College. He is an honorary doctor at Universidad Carlos III de Madrid. **www.stroustrup.com**.

In other words, I have serious industrial and academic experience.

I used earlier versions of this book to teach thousands of first-year university students, many of whom had never written a line of code in their lives. Beyond that, I have taught people of all levels from undergraduates to seasoned developers and scientists. I currently teach final-year undergraduates and grad students at Columbia University.

I do have a life outside work. I'm married with two children and five grandchildren. I read a lot, including history, science fiction, crime, and current affairs. I like most kinds of music, including classical, classical rock, blues, and country. Good food with friends is essential and I enjoy visiting interesting places all over the world. To be able to enjoy the good food, I run.

For more biographical information, see **www.stroustrup.com/bio.html**.

0.6 Bibliography

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

[ARM] M. Ellis and B. Stroustrup: *The Annotated C++ Reference Manual* Addison Wesley. 1990. ISBN 0-201-51459-1.

[AS,2009] Alexander Stepanov and Paul McJones: *Elements of Programming*. Addison-Wesley. 2009. ISBN 978-0-321-63537-2.

[AW,2019] Anthony Williams: *C++ Concurrency in Action: Practical Multithreading (Second edition)*. Manning Publishing. 2019. ISBN 978-1617294693.

[BS,2022] B. Stroustrup: *A Tour of C++ (3rd edition)*. Addison-Wesley, 2022. ISBN 978-0136816485.

[CG] B. Stroustrup and H. Sutter: *C++ Core Guidelines*. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md.

[C++20] Richard Smith (editor): *The C++ Standard*. ISO/IEC 14882:2020.

[DnE] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.

[HOPL-2] B. Stroustrup: *A History of C++: 1979–1991*. Proc. ACM History of Programming Languages Conference (HOPL-2). ACM Sigplan Notices. Vol 28, No 3. 1993.

[HOPL-3] B. Stroustrup: *Evolving a language in and for the real world: C++ 1991-2006*. ACM HOPL-III. June 2007.

[HOPL-4] B. Stroustrup: *Thriving in a crowded and changing world: C++ 2006-2020*. ACM/SIGPLAN History of Programming Languages conference, HOPL-IV. June 2021.

[K&R] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*. Prentice-Hall. 1978. ISBN 978-0131101630.

[Simula] Graham Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard: *SIMULA BEGIN*. Studentlitteratur. 1979. ISBN 91-44-06212-5.

[TC++PL] B. Stroustrup: *The C++ Programming Language (Fourth Edition)*. Addison-Wesley, 2013. ISBN 0321563840.

Postscript

Each chapter provides 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 many thousands of programmers have found stimulating and fun.

Part I
The Basics

Part I 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.

This page intentionally left blank

1

Hello, World!

Programming is learned
by writing programs.
– Brian Kernighan

Here, we present the simplest C++ program that actually does anything. The purpose of writing this program is to

- Let you try your programming environment
- Give you a first feel of how you can get a computer to do things for you

Thus, we present the notion of a program, the idea of translating a program from human-readable form to machine instructions using a compiler, and finally executing those machine instructions.

1.1 Programs

To get a computer to do something, you (or someone else) have to tell it exactly – in excruciating detail – what to do. Such a description of "what to do" is called a *program*, and *programming* is the activity of writing and testing such programs.

In a sense, we have all programmed before. After all, we have given descriptions of tasks to be done, such as "how to drive to the nearest cinema," "how to find the upstairs bathroom," and "how to heat a meal in the microwave." The difference between such descriptions and programs is one of degree of precision: humans tend to compensate for poor instructions by using common sense, but computers don't. For example, "turn right in the corridor, up the stairs, it'll be on your left" is probably a fine description of how to get to the upstairs bathroom. However, when you look at those simple instructions, you'll find the grammar sloppy and the instructions incomplete. A human easily compensates. For example, assume that you are sitting at the table and ask for directions to the bathroom. You don't need to be told to get up from your chair to get to the corridor, somehow walk around (and not across or under) the table, not to step on the cat, etc. You'll not have to be told not to bring your knife and fork or to remember to switch on the light so that you can see the stairs. Opening the door to the bathroom before entering is probably also something you don't have to be told.

In contrast, computers are *really* dumb. They have to have everything described precisely and in detail. Consider again "turn right in the corridor, up the stairs, it'll be on your left." Where is the corridor? What's a corridor? What is "turn right"? What stairs? How do I go up stairs? (One step at a time? Two steps? Slide up the banister?) What is on my left? When will it be on my left? To be able to describe "things" precisely for a computer, we need a precisely defined language with a specific grammar (English is far too loosely structured for that) and a well-defined vocabulary for the kinds of actions we want performed. Such a language is called a *programming language*, and C++ is a programming language designed for a wide selection of programming tasks. When a computer can perform a complex task given simple instructions, it is because someone has taught it to do so by providing a program.

If you want greater philosophical detail about computers, programs, and programming, see PPP2.Ch1 and PPP2.Ch22. Here, we start with a very simple program and the tools and techniques you need to get it to run.

1.2 The classic first program

Here is a version of the classic first program. It writes "Hello, World!" on your screen:

```
// This program outputs the message "Hello, World!" to the monitor

import std;              // gain access to the C++ standard library

int main()               // C++ programs start by executing the function main
{
        std::cout << "Hello, World!\n";          // output "Hello, World!"
        return 0;
}
```

Think of this text as a set of instructions that we give to the computer to execute, much as we would give a recipe to a cook to follow, or as a list of assembly instructions for us to follow to get a new toy working. Let's discuss what each line of this program does, starting with the line

```
std::cout << "Hello, World!\n";                    // output "Hello, World!"
```

That's the line that actually produces the output. It prints the characters **Hello, World!** followed by a newline; that is, after writing **Hello, World!**, the cursor will be placed at the start of the next line. A *cursor* is a little blinking character or line showing where you can type the next character. CC

In C++, string literals are delimited by double quotes ("); that is, **"Hello, World!\n"** is a string of characters. The **\n** is a *special character* indicating a newline. The name **cout** refers to a standard output stream. Characters "put into **cout**" using the output operator **<<** will appear on the screen. The name **cout** is pronounced "see-out" and is an abbreviation of "**c**haracter **out**put stream." You'll find abbreviations rather common in programming. Naturally, an abbreviation can be a bit of a nuisance the first time you see it and have to remember it, but once you start using abbreviations repeatedly, they become second nature, and they are essential for keeping program text short and manageable.

The **std::** in **std::cout** says that the **cout** is to be found in the standard library that we made accessible with **import std;**

The end of that line

```
// output "Hello, World!"
```

is a comment. Anything written after the token // (that's the character /, called "slash," twice) on a line is a comment. Comments are ignored by the compiler and written for the benefit of programmers who read the code. Here, we used the comment to tell you what the beginning of that line actually did.

Comments are written to describe what the program is intended to do and in general to provide information useful for humans that can't be directly expressed in code. The person most likely to benefit from the comments in your code is you – when you come back to that code next week, or next year, and have forgotten exactly why you wrote the code the way you did. So, document your programs well. In §4.7.2.1 and §6.6.4, we'll discuss what makes good comments.

A program is written for two audiences. Naturally, we write code for computers to execute. However, we spend long hours reading and modifying the code. Thus, programmers are another audience for programs. So, writing code is also a form of human-to-human communication. In fact, it makes sense to consider the human readers of our code our primary audience: if they (we) don't find the code reasonably easy to understand, the code is unlikely to ever become correct. So, please don't forget: code is for reading – do all you can to make it readable. CC

The first line of the program is a typical comment; it simply tells the human reader what the program is supposed to do:

```
// This program outputs the message "Hello, World!" to the monitor
```

Such comments are useful because the code itself says what the program does, not what we meant it to do. Also, we can usually explain (roughly) what a program should do to a human much more concisely than we can express it (in detail) in code to a computer. Often such a comment is the first part of the program we write. If nothing else, it reminds us what we are trying to do.

The next line

```
import std;
```

is a module import statement. It instructs the computer to make available ("to import") facilities from a module called **std**. This is a standard module making all facilities from the C++ standard library available. We will explain its contents as we go along. For this program, the importance of **std** is that we make the standard C++ stream I/O facilities available. Here, we just use the standard output stream, **cout**, and its output operator, **<<**.

How does a computer know where to start executing a program? It looks for a function called **main** and starts executing the instructions it finds there. Here is the function **main** of our "Hello, World!" program:

```
int main()      // C++ programs start by executing the function main
{
        std::cout << "Hello, World!\n";          // output "Hello, World!"
        return 0;
}
```

CC Every C++ program must have a function called **main** to tell it where to start executing. A function is basically a named sequence of instructions for the computer to execute in the order in which they are written. A function has four parts:

- A *return type*, here **int** (meaning "integer"), which specifies what kind of result, if any, the function will return to whoever asked for it to be executed. The word **int** is a reserved word in C++ (a *keyword*), so **int** cannot be used as the name of anything else.
- A *name*, here **main**.
- A *parameter list* enclosed in parentheses (see §7.2 and §7.4), here **()**; in this case, the parameter list is empty.
- A *function body* enclosed in a set of "curly braces," **{ }**, which lists the actions (called *statements*) that the function is to perform.

It follows that the minimal C++ program is simply

```
int main() { }
```

That's not of much use, though, because it doesn't do anything. The **main()** ("the main function") of our "Hello, World!" program has two statements in its body:

```
std::cout << "Hello, World!\n";                    // output "Hello, World!"
return 0;
```

First it'll write **Hello, World!** to the screen, and then it will return a value **0** (zero) to whoever called it. Since **main()** is called by "the system," we won't use that return value. However, on some systems (notably Unix/Linux) it can be used to check whether the program succeeded. A zero (**0**) returned by **main()** indicates that the program terminated successfully.

A part of a C++ program that specifies an action is called a *statement*.

1.3 Compilation

CC

C++ is a compiled language. That means that to get a program to run, you must first translate it from the human-readable form to something a machine can "understand." That translation is done by a program called a *compiler*. What you read and write is called *source code* or *program text*, and what the computer executes is called *object code* or *machine code*. Typically, C++ source code files are given the suffix **.cpp** (e.g., **hello_world.cpp**) and object code files are given the suffix **.obj** (on Windows) or **.o** (Linux). The plain word *code* is therefore ambiguous and can cause confusion; use it with care only when it is obvious what's meant by it. Unless otherwise specified, we use *code* to mean "source code" or even "the source code except the comments," because comments really are there just for us humans and are not seen by the compiler generating object code.

The compiler reads your source code and tries to make sense of what you wrote. It looks to see if your program is grammatically correct, if every word has a defined meaning, and if there is anything obviously wrong that can be detected without trying to actually execute the program. You'll find that compilers are rather picky about syntax. Leaving out any detail of our program, such as **import**ing a **module** file, a semicolon, or a curly brace, will cause errors. Similarly, the compiler has absolutely zero tolerance for spelling mistakes. Let us illustrate this with a series of examples, each of which has a single small error. Each error is an example of a kind of mistake we often make:

```
int main()
{
        std::cout << "Hello, World!\n";
        return 0;
}
```

We didn't provide the compiler with anything to explain what **std::cout** was, so the compiler complains. To correct that, let's add the **import**:

```
import std;
int main()
{
        cout << "Hello, World!\n";
        return 0;
}
```

The compiler again complains: We made the standard library available but forgot to tell the compiler to look in **std** for **cout**. The compiler also objects to this:

```
import std;
int main()
{
        std::cout << "Hello, World!\n;
        return 0;
}
```

First we make the (symbol,price) map:

```
map<string,double> dow_price = {  // Dow Jones Industrial index (symbol,price);
                                  // for up-to-date quotes see www.djindexes.com
        {"MMM",104.48},
        {"AAPL",165.02},
        {"MSFT",285.76},
        // ...
};
```

The (symbol,weight) map:

```
map<string,double> dow_weight = {       // Dow (symbol,weight)
        {"MMM", 2.41},
        {"AAPL",2.84},
        {"MSFT",4.88},
        // ...
};
```

The (symbol,name) map:

```
map<string,string> dow_name = {   // Dow (symbol,name)
        {"MMM","3M"},
        {"AAPL","Apple"},
        {"MSFT","Microsoft"},
        // ...
};
```

Given those maps, we can conveniently extract all kinds of information. For example:

```
double caterpillar = dow_price ["CAT"];              // read values from a map
double boeing_price = dow_price ["BA"];

if (dow_price.find("INTC") != dow_price.end())       // find an entry in a map
        cout << "Intel is in the Dow\n";
```

Iterating through a map is easy:

```
for (const auto& [symbol,price] : dow_price)
        cout << symbol << '\t' << price << '\t' << dow_name[symbol] << '\n';
```

AA Why might someone keep such data in **map**s rather than **vector**s? We used a **map** to make the association between the different values explicit. That's one common reason. Another is that a **map** keeps its elements in the order defined by its key. When we iterated through **dow** above, we output the symbols in alphabetical order; had we used a **vector** we would have had to sort. The most common reason to use a **map** is simply that we want to look up values based on the key. For large sequences, finding something using **find()** is far slower than looking it up in a sorted structure, such as a **map**.

> TRY THIS
>
> Get this little example to work. Then add a few companies of your own choice, with weights of your choice.

I

Index

Knowledge is of two kinds.
We know a subject ourselves,
or we know where
we can find information on it.
– Samuel Johnson

Token

A

B

C

D

E

F

G

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z