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).

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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 U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2024932369

Copyright 2024 by Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

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 ix

0 Notes to the Reader 1

0.1 The structure of this book ... 2
0.2 A philosophy of teaching and learning ... 5
0.3 ISO standard C++ ... 8
0.4 PPP support ... 11
0.5 Author biography ... 13
0.6 Bibliography ... 13

Part I: The Basics

1 Hello, World! 17

1.1 Programs ... 18
1.2 The classic first program ... 18
1.3 Compilation ... 21
1.4 Linking ... 23
1.5 Programming environments ... 24

2 Objects, Types, and Values 29

2.1 Input ... 30
2.2 Variables ... 32
2.3 Input and type ... 33
2.4 Operations and operators ... 34
2.5 Assignment and initialization ... 36
2.6 Names ... 40
2.7 Types and objects ... 42
2.8 Type safety ... 43
2.9 Conversions ... 44
2.10 Type deduction: auto ... 46

3 Computation 51

3.1 Computation ... 52
3.2 Objectives and tools ... 53
3.3 Expressions ... 55
3.4 Statements ... 58
3.5 Functions ... 68
3.6 vector ... 71
3.7 Language features ... 77

4 Errors! 83

4.1 Introduction ... 84
4.2 Sources of errors ... 85
4.3 Compile-time errors ... 86
4.4 Link-time errors ... 88
4.5 Run-time errors ... 89
4.6 Exceptions ... 94
4.7 Avoiding and finding errors ... 99

5 Writing a Program 115

5.1 A problem ... 116
5.2 Thinking about the problem ... 116
5.3 Back to the calculator! ... 119
5.4 Back to the drawing board ... 126
5.5 Turning a grammar into code ... 130
5.6 Trying the first version ... 136
5.7 Trying the second version ... 140
5.8 Token streams ... 142
5.9 Program structure ... 146

6 Completing a Program 151

6.1 Introduction 152
6.2 Input and output 152
6.3 Error handling 154
6.4 Negative numbers 156
6.5 Remainder: % 157
6.6 Cleaning up the code 158
6.7 Recovering from errors 164
6.8 Variables 167

7 Technicalities: Functions, etc. 179

7.1 Technicalities 180
7.2 Declarations and definitions 181
7.3 Scope 186
7.4 Function call and return 190
7.5 Order of evaluation 206
7.6 Namespaces 209
7.7 Modules and headers 211

8 Technicalities: Classes, etc. 221

8.1 User-defined types 222
8.2 Classes and members 223
8.3 Interface and implementation 223
8.4 Evolving a class: Date 225
8.5 Enumerations 233
8.6 Operator overloading 236
8.7 Class interfaces 237

Part II: Input and Output

9 Input and Output Streams 251

9.1 Input and output 252
9.2 The I/O stream model 253
9.3 Files 254
9.4 I/O error handling 258
9.5 Reading a single value 261
9.6 User-defined output operators 266
9.7 User-defined input operators 266
9.8 A standard input loop 267

9.9 Reading a structured file 269
9.10 Formatting 276
9.11 String streams 283

10 A Display Model 289

10.1 Why graphics? 290
10.2 A display model 290
10.3 A first example 292
10.4 Using a GUI library 295
10.5 Coordinates 296
10.6 Shapes 297
10.7 Using Shape primitives 297
10.8 Getting the first example to run 309

11 Graphics Classes 315

11.1 Overview of graphics classes 316
11.2 Point and Line 317
11.3 Lines 320
11.4 Color 323
11.5 Line_style 325
11.6 Polylines 328
11.7 Closed shapes 333
11.8 Text 346
11.9 Mark 348
11.10 Image 350

12 Class Design 355

12.1 Design principles 356
12.2 Shape 360
12.3 Base and derived classes 367
12.4 Other Shape functions 375
12.5 Benefits of object-oriented programming 376

13 Graphing Functions and Data 381

13.1 Introduction 382
13.2 Graphing simple functions 382
13.3 Function 386
13.4 Axis 390

13.5 Approximation 392
13.6 Graphing data 397

14 Graphical User Interfaces **409**

14.1 User-interface alternatives 410
14.2 The “Next” button 411
14.3 A simple window 412
14.4 Button and other Widgets 414
14.5 An example: drawing lines 419
14.6 Simple animation 426
14.7 Debugging GUI code 427

Part III: Data and Algorithms

15 Vector and Free Store **435**

15.1 Introduction 436
15.2 vector basics 437
15.3 Memory, addresses, and pointers 439
15.4 Free store and pointers 442
15.5 Destructors 447
15.6 Access to elements 451
15.7 An example: lists 452
15.8 The this pointer 456

16 Arrays, Pointers, and References **463**

16.1 Arrays 464
16.2 Pointers and references 468
16.3 C-style strings 471
16.4 Alternatives to pointer use 472
16.5 An example: palindromes 475

17 Essential Operations **483**

17.1 Introduction 484
17.2 Access to elements 484
17.3 List initialization 486
17.4 Copying and moving 488
17.5 Essential operations 495

17.6 Other useful operations 500
17.7 Remaining Vector problems 502
17.8 Changing size 504
17.9 Our Vector so far 509

18 Templates and Exceptions **513**

18.1 Templates 514
18.2 Generalizing Vector 522
18.3 Range checking and exceptions 525
18.4 Resources and exceptions 529
18.5 Resource-management pointers 537

19 Containers and Iterators **545**

19.1 Storing and processing data 546
19.2 Sequences and iterators 552
19.3 Linked lists 555
19.4 Generalizing Vector yet again 560
19.5 An example: a simple text editor 566
19.6 vector, list, and string 572

20 Maps and Sets **577**

20.1 Associative containers 578
20.2 map 578
20.3 unordered_map 585
20.4 Timing 586
20.5 set 589
20.6 Container overview 591
20.7 Ranges and iterators 597

21 Algorithms **603**

21.1 Standard-library algorithms 604
21.2 Function objects 610
21.3 Numerical algorithms 614
21.4 Copying 619
21.5 Sorting and searching 620

Index **625**

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 nontrivial 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
General approach; Drills, exercises, etc.; What comes after this book?
§0.2 A philosophy of teaching and learning
A note to students; A note to teachers
§0.3 ISO standard C++
Portability; Guarantees; A brief history of C++
§0.4 PPP support
Web resources
§0.5 Author biography
§0.6 Bibliography

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.

CC

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.

0.1.2 Drills, exercises, etc.

AA

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

- *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

AA

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.

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:

- `module`s (§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.

Chapter 1: Hello, World!
Chapter 2: Objects, Types, and Values
Chapter 3: Computation
Chapter 4: Errors!
Chapter 5: Writing a Program
Chapter 6: Completing a Program
Chapter 7: Technicalities: Functions, etc.
Chapter 8: Technicalities: Classes, etc.

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
§1.2 The classic first program
§1.3 Compilation
§1.4 Linking
§1.5 Programming environments

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!"
```

CC

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.

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.

CC

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.

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;
}
```

We didn't terminate the string with a ". The compiler also objects to this:

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

The abbreviation int is used in C++ rather than the word integer. The compiler doesn't like this:

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

We used < (the less-than operator) rather than << (the output operator). Another error:

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

We used single quotes rather than double quotes to delimit the string. Finally, the compiler gives an error for this:

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

We forgot to terminate the output statement with a semicolon. Note that many C++ statements are terminated by a semicolon (;). The compiler needs those semicolons to know where one statement ends and the next begins. There is no really short, fully correct, and nontechnical way of summarizing where semicolons are needed. For now, just copy our pattern of use, which can be summarized as: "Put a semicolon after every expression that doesn't end with a right curly brace (})."

Finally, let's try something that surprisingly works:

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

For historical reasons, we can leave out the return statement in main (and only in main) and it is as if we had written return 0; and the end of main's body to indicate successful completion.

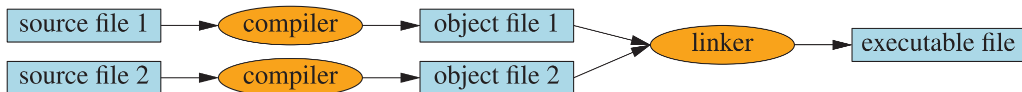
Why do we spend two pages of good space and minutes of your precious time showing you examples of trivial errors in a trivial program? To make the point that you – like all programmers – will spend a lot of time looking for errors in program source text. Most of the time, we look at text with errors in it. After all, if we were convinced that some code was correct, we'd be looking at some other code or taking the time off. It came as a major surprise to the brilliant early computer pioneers that they were making mistakes and had to devote a major portion of their time to finding them. It is still a surprise to most newcomers to programming.

When you program, you'll get quite annoyed with the compiler at times. Sometimes it appears to complain about unimportant details (such as a missing semicolon) or about things you consider "obviously right." However, the compiler is usually right: when it gives an error message and refuses to produce object code from your source code, there is something not quite right with your program; that is, the meaning of what you wrote isn't precisely defined by the C++ standard.

The compiler has no common sense (it isn't human) and is very picky about details. Since it has no common sense, you wouldn't like it to try to guess what you meant by something that "looked OK" but didn't conform to the definition of C++. If it did and its guess was different from yours, you could end up spending a lot of time trying to figure out why the program didn't do what you thought you had told it to do. When all is said and done, the compiler saves us from a lot of self-inflicted problems. It saves us from many more problems than it causes. So, please remember: the compiler is your friend; possibly, the compiler is the best friend you have when you program.

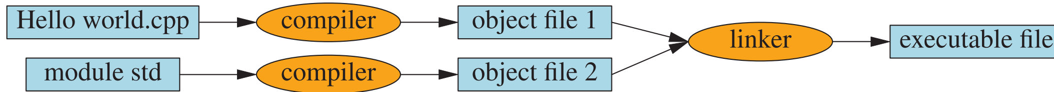
1.4 Linking

A program usually consists of several separate parts, often developed by different people. For example, the "Hello, World!" program consists of the part we wrote plus parts of the C++ standard library. These separate parts (sometimes called *modules* or *translation units*) must be compiled and the resulting object code files must be linked together to form an executable program. The program that links such parts together is (unsurprisingly) called a *linker*:



The output from a linker is called an *executable file* and on Windows its name is often given the suffix **.exe**. Please note that object code and executables are *not* portable among systems. For example, when you compile for a Windows machine, you get object code for Windows that will not run on a Linux machine.

A *library* is simply some code – usually written by others – that we access using declarations found in an **import**ed **module**. For example:



A *declaration* is a program statement specifying how a piece of code can be used; we'll examine declarations in detail later (§3.5.2, §7.2).

Errors found by the compiler are called *compile-time errors*, errors found by the linker are called *link-time errors*, and errors not found until the program is run are called *run-time errors* or *logic errors*. Generally, compile-time errors are easier to understand and fix than link-time errors, and link-time errors are often easier to find and fix than run-time errors. In §4, we discuss errors and the ways of handling them in greater detail.

1.5 Programming environments

To program, we use a programming language. We also use a compiler to translate our source code into object code and a linker to link our object code into an executable program. In addition, we use some program to enter our source code text into the computer and to edit it. These are just the first and most crucial tools that constitute our programmer's tool chest or "program development environment."

If you work from a command-line window, as many professional programmers do, you will have to issue the compile and link commands yourself (§0.3.1). If instead you use an IDE ("interactive development environment" or "integrated development environment"), as many professional programmers also do, a simple click on the correct button will do the job.

IDEs usually include an editor with helpful features like color coding to help distinguish between comments, keywords, and other parts of your program source code, plus other facilities to help you debug your code, compile it, and run it. *Debugging* is the activity of finding errors in a program and removing them; you'll hear a lot about that along the way. An error in a program is often called a *bug*, hence the term "debugging." The reason for calling an error "a bug" is that in a very early system a program failed because an insect had found its way into the computer PPP2.§22.2.2.2.

Working with this book, you can use any system that provides an up-to-date, standards-conforming implementation of C++. Most of what we say will be true for all implementations of C++, and the code will run everywhere. In our work, we use several different implementations.

Drill

So far, we have talked about programming, code, and tools (such as compilers). Now you have to get a program to run. This is a crucial point in this book and an important step in learning to program. This is where you start to develop practical skills and good programming habits. The exercises for this chapter are focused on getting you acquainted with your software development environment. Once you get the "Hello, World!" program to run, you will have passed the first major milestone as a programmer.

AA

The purpose of a drill is to establish or reinforce your practical programming skills and give you experience with programming environment tools. Typically, a drill is a sequence of modifications to a single program, "growing" it from something completely trivial to something that might be a useful part of a real program. A traditional set of exercises is designed to test your initiative, cleverness, or inventiveness. In contrast, a drill requires little invention from you. Typically, sequencing is crucial, and each individual step should be easy (or even trivial). Please don't try to be clever and skip steps; on average that will slow you down or even confuse you.

You might think you understand everything you read and everything your Mentor or instructor told you, but repetition and practice are necessary to develop programming skills. In this regard, programming is like athletics, music, dance, or any skill-based craft. Imagine people trying to compete in any of those fields without regular practice. You know how well they would perform. Constant practice – for professionals that means lifelong constant practice – is the only way to develop and maintain a high-level practical skill.

XX

So, never skip the drills, no matter how tempted you are; they are essential to the learning process. Just start with the first step and proceed, testing each step as you go to make sure you are doing it right.

AA

Don't be alarmed if you don't understand every detail of the syntax you are using, and don't be afraid to ask for help from instructors or friends. Keep going, do all of the drills and many of the exercises, and all will become clear in due time.

So, here is your first drill:

[1] Figure out how to compile and run a program on your machine. This may be a good time to get a bit of help from someone who has done that before.

[2] If you use an IDE, set up an empty console C++ project called **hello_world**. Otherwise, if you plan to use the command line, get a command window, figure out how to use an editor to type in your code, and see §0.3.1.

[3] Type in **hello_world.cpp**, exactly as specified below, save it in your practice directory (folder), and include it in your hello_world project.

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

What if you don't have an up-to-date C++ implementation that supports module **std**? Then, use a less elegant and less efficient technique that has worked since the early days of C: **#include** a *header file* as is explained in §7.7.2:

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

[4] Compile and run the "Hello, World!" program. An IDE will have a compile-and-run button. Even something as simple as "Hello, World!" very rarely compiles and runs in a first attempt to use a new programming language or a new programming environment.

Find the problem and fix it! This is a point where asking for help from a more experienced person is sensible, but be sure to understand what you are shown so that you can do it all by yourself before proceeding further.

[5] By now, you have probably encountered some errors and had to correct them. Now is the time to get a bit better acquainted with your compiler's error-detection and error-reporting facilities! Try the eight programs from §1.3 to see how your programming environment reacts. Think of at least five more errors you might have made typing in your program (e.g., leave the Caps Lock key on while typing a word, or type a comma instead of a semicolon) and try each to see what happens when you try to compile and run those versions.

Review

AA

The basic idea of these review questions is to give you a chance to see if you have noticed and understood the key points of the chapter. You may have to refer back to the text to answer a question; that's normal and expected. You may have to reread whole sections; that too is normal and expected. However, if you have to reread the whole chapter or have problems with every review question, you should consider whether your style of learning is effective. Are you reading too fast? Did you follow some of the "Try this" suggestions? Should you study with a friend so that you can discuss problems with the explanations in the text?

[1] What is the purpose of the "Hello, World!" program?
[2] Name the four parts of a function.
[3] Name a function that must appear in every C++ program.
[4] In the "Hello, World!" program, what is the purpose of the line `return 0;`?
[5] What is the purpose of the compiler?
[6] What is the purpose of the `import` statement?
[7] What is the purpose of the `#include` directive?
[8] What does a `.cpp` suffix at the end of a file name signify in C++?
[9] What does the linker do for your program?
[10] What is the difference between a source file and an object file?
[11] What is an executable?
[12] What is an IDE and what does it do for you?
[13] How do you get a compiled program to run?
[14] What is a comment?
[15] What is the purpose of a drill?
[16] If you understand everything in the textbook, why is it necessary to practice?

Most review questions have a clear answer in the chapter in which they appear. However, we do occasionally include questions to remind you of relevant information from other chapters and sometimes even relating to the world outside this book. We consider that fair; there is more to writing good software and thinking about the implications of doing so than fits into an individual chapter or book.

Terms

These terms present 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.

//	executable	main()	<<
function	object code	C++	header file
output	comment	IDE	program
compiler	import	source code	compile-time
error	library	statement	cout
linker	module	#include	std
command line	bug	debugging	

You might like to gradually develop a glossary written in your own words. You can do that by repeating exercise 5 below for each chapter.

Exercises

We list drills separately from exercises; always complete the chapter drill before attempting an exercise. Doing so will save you time.

[1] Change the program to output the two lines

```
Hello, programming!
Here we go!
```

[2] Expanding on what you have learned, write a program that lists the instructions for a computer to find the upstairs bathroom, discussed in §1.1. Can you think of any more steps that a person would assume, but that a computer would not? Add them to your list. This is a good start in "thinking like a computer." Warning: For most people, "go to the bathroom" is a perfectly adequate instruction. For someone with no experience with houses or bathrooms (imagine a stone-age person, somehow transported into your dining room) the list of necessary instructions could be *very* long. Please don't use more than a page. For the benefit of the reader, you may add a short description of the layout of the house you are imagining.

[3] Write a description of how to get from the front door of your dorm room, apartment, house, whatever, to the door of your classroom (assuming you are attending some school; if you are not, pick another target). Have a friend try to follow the instructions and annotate them with improvements as he or she goes along. To keep friends, it may be a good idea to "field test" those instructions yourself before giving them to a friend.

[4] Find a good cookbook. Read the instructions for baking blueberry muffins (if you are in a country where "blueberry muffins" is a strange, exotic dish, use a more familiar dish instead). Please note that with a bit of help and instruction, most of the people in the world can bake delicious blueberry muffins. It is not considered advanced or difficult fine cooking. However, for the author, few exercises in this book are as difficult as this one. It is amazing what you can do with a bit of practice.

- Rewrite those instructions so that each individual action is in its own numbered paragraph. Be careful to list all ingredients and all kitchen utensils used at each step. Be

careful about crucial details, such as the desired oven temperature, preheating the oven, the preparation of the muffin pan, the way to time the cooking, and the need to protect your hands when removing the muffins from the oven.

- Consider those instructions from the point of view of a cooking novice (if you are not one, get help from a friend who does not know how to cook). Fill in the steps that the book's author (almost certainly an experienced cook) left out for being obvious.
- Build a glossary of terms used. (What's a muffin pan? What does preheating do? What do you mean by "oven"?)
- Now bake some muffins and enjoy your results.

[5] Write a definition for each of the terms from "Terms." First try to see if you can do it without looking at the chapter (not likely), then look through the chapter to find definitions. You might find the difference between your first attempt and the book's version interesting. You might consult some suitable online glossary, such as www.stroustrup.com/glossary.html. By writing your own definition before looking it up, you reinforce the learning you achieved through your reading. If you have to reread a section to form a definition, that just helps you to understand. Feel free to use your own words for the definitions and make the definitions as detailed as you think reasonable. Often, an example after the main definition will be helpful. You may like to store the definitions in a file so that you can add to them from the "Terms" sections of later chapters.

Postscript

CC

What's so important about the "Hello, World!" program? Its purpose is to get us acquainted with the basic tools of programming. We tend to do an extremely simple example, such as "Hello, World!" whenever we approach a new tool. That way, we separate our learning into two parts: first we learn the basics of our tools with a trivial program, and later we learn about more complicated programs without being distracted by our tools. Learning the tools and the language simultaneously is far harder than doing first one and then the other. This approach to simplifying learning a complex task by breaking it into a series of small (and more manageable) steps is not limited to programming and computers. It is common and useful in most areas of life, especially in those that involve some practical skill.

2

Objects, Types, and Values

Fortune favors the prepared mind.
– Louis Pasteur

This chapter introduces the basics of storing and using data in a program. To do so, we first concentrate on reading in data from the keyboard. After establishing the fundamental notions of objects, types, values, and variables, we introduce several operators and give many examples of use of variables of types **char**, **int**, **double**, and **string**.

§2.1 Input
§2.2 Variables
§2.3 Input and type
§2.4 Operations and operators
§2.5 Assignment and initialization
An example: detect repeated words; Composite assignment operators; An example: find repeated words
§2.6 Names
§2.7 Types and objects
§2.8 Type safety
§2.9 Conversions
§2.10 Type deduction: **auto**

2.1 Input

The “Hello, World!” program just writes to the screen. It produces output. It does not read anything; it does not get input from its user. That’s rather a bore. Real programs tend to produce results based on some input we give them, rather than just doing exactly the same thing each time we execute them.

CC

To read something, we need somewhere to read into; that is, we need somewhere in the computer’s memory to place what we read. We call such a “place” an object. An *object* is a region of memory with a *type* that specifies what kind of information can be placed in it. A named object is called a *variable*. For example, character strings are put into **string** variables and integers are put into **int** variables. You can think of an object as a “box” into which you can put a value of the object’s type:

int:
age: 42

This would represent an object of type **int** named **age** containing the integer value **42**. Using a string variable, we can read a string from input and write it out again like this:

```
// read and write a first name
#include "PPP.h"

int main()
{
        cout << "Please enter your first name (followed by "enter"):\n";
        string first_name;              // first_name is a variable of type string
        cin >> first_name;              // read characters into first_name
        cout << "Hello, " << first_name << "!\n";
}
```

The **#include** and the **main()** are familiar from Chapter 1. Since the **#include** or the equivalent direct use of **import** is needed for all our programs, we’ll leave it out of our presentation to avoid distraction. Similarly, we’ll sometimes present code that will work only if it is placed in **main()** or some other function, like this:

```
cout << "Please enter your first name (followed by 'enter'):\n";
```

We assume that you can figure out how to put such code into a complete program for testing.

The first line of **main()** simply writes out a message encouraging the user to enter a first name. Such a message is typically called a *prompt* because it prompts the user to take an action. The next lines define a variable of type **string** called **first_name**, read input from the keyboard into that variable and write out a greeting. Let’s look at those three lines in turn:

```
string first_name;    // first_name is a variable of type string
```

This sets aside an area of memory for holding a string of characters and gives it the name **first_name**:

string:
first_name: [ ]

A statement that introduces a new name into a program and sets aside memory for a variable is called a *definition*.

The next line reads characters from input (the keyboard) into that variable:

```
cin >> first_name;    // read characters into first_name
```

The name `cin` refers to the standard input stream (pronounced "see-in," for "character input") defined in the standard library. The second operand of the `>>` operator ("get from") specifies where that input goes. So, if we type some first name, say `Nicholas`, followed by a newline, the string `"Nicholas"` becomes the value of `first_name`:

string:
first_name: [Nicholas]

AA

The newline is necessary to get the machine's attention. Until a newline is entered (the Enter key is hit), the computer simply collects characters. That "delay" gives you the chance to change your mind, erase some characters and replace them with others before hitting Enter. The newline will not be part of the string stored in memory.

Having gotten the input string into `first_name`, we can use it:

```
cout << "Hello, " << first_name << "!\n";
```

This prints `Hello,` followed by `Nicholas` (the value of `first_name`) followed by `!` and a newline (`'\n'`) on the screen:

```
Hello, Nicholas!
```

If we had liked repetition and extra typing, we could have written three separate statements instead:

```
cout << "Hello, ";
cout << first_name;
cout << "!\n";
```

However, we are indifferent typists, and – more importantly – strongly dislike needless repetition (because repetition provides opportunity for errors), so we combined those three output operations into a single statement.

Note the way we use quotes around the characters in `"Hello, "` but not for `first_name`. We use quotes when we want a literal string. When we don't quote, we refer to the value of something with a name. Consider:

```
cout << "first_name" << " is " << first_name;
```

Here, `"first_name"` gives us the ten characters `first_name` and plain `first_name` gives us the value of the variable `first_name`, in this case, `Nicholas`. So, we get

```
first_name is Nicholas
```

2.2 Variables

CC

Basically, we can do nothing of interest with a computer without storing data in memory, the way we did it with the input string in the example above. The "places" in which we store data are called *objects*. To access an object, we need a *name*. A named object is called a *variable* and has a specific *type* (such as **int** or **string**) that determines what can be put into the object (e.g., **123** can go into an **int** and **"Hello, World!\n"** can go into a **string**) and which operations can be applied (e.g., we can multiply **int**s using the * operator and compare **string**s using the **<=** operator). The data items we put into variables are called *values*. A statement that defines a variable is (unsurprisingly) called a *definition*, and a definition can (and usually should) provide an initial value. Consider:

```
string name = "Annemarie";
int number_of_steps = 39;
```

The value after the {**=**} is called an *initializer*.

You can visualize these variables like this:

You cannot put values of the wrong type into a variable:

```
string name2 = 39;                  // error: 39 isn't a string
int number_of_steps = "Annemarie";  // error: "Annemarie" is not an int
```

The compiler remembers the type of each variable and makes sure that you use it according to its type, as specified in its definition.

C++ provides a rather large number of types. You can find complete lists on the Web (e.g., **cppreference.com**). However, you can write perfectly good programs using only five of those:

```
int number_of_steps = 39;       // int for integers
double flying_time = 3.5;       // double for floating-point numbers
char decimal_point = '.';       // char for individual characters
string name = "Annemarie";      // string for character strings
bool tap_on = true;             // bool for logical variables
```

The reason for the name **double** is historical: **double** is short for "double-precision floating point." Floating point is the computer's approximation to the mathematical concept of a real number.

Note that each of these types has its own characteristic style of literals:

```
39              // int: an integer
3.5             // double: a floating-point number
'.'             // char: an individual character enclosed in single quotes
"Annemarie"     // string: a sequence of characters delimited by double quotes
true            // bool: either true or false
```

That is, a sequence of digits (such as **1234**, **2**, or **976**) denotes an integer, a single character in single quotes (such as **'1'**, **'@'**, or **'x'**) denotes a character, a sequence of digits with a decimal point (such as **1.234**, **0.12**, or **.98**) denotes a floating-point value, and a sequence of characters enclosed in double quotes (such as **"1234"**, **"Howdy!"**, or **"Annemarie"**) denotes a string.

2.3 Input and type

The input operation >> ("get from") is sensitive to type; that is, it reads according to the type of variable you read into. For example: CC

```
int main()        // read name and age
{
        cout << "Please enter your first name and age\n";
        string first_name = "???";          // string variable  ("???" indicates "don't know the name")
        int age = -1;                       // integer variable (-1 means "don't know the age")
        cin >> first_name >> age;           // read a string followed by an integer
        cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

So, if you type in Carlos 22 the >> operator will read Carlos into first_name, 22 into age, and produce this output:

```
Hello, Carlos (age 22)
```

Why won't it read (all of) Carlos 22 into first_name? Because, by convention, reading of strings is terminated by what is called *whitespace*, that is, space, newline, and tab characters. Otherwise, whitespace by default is ignored by >>. For example, you can add as many spaces as you like before a number to be read; >> will just skip past them and read the number.

Just as we can write several values in a single output statement, we can read several values in a single input statement. Note that << is sensitive to type, just as >> is, so we can output the int variable age as well as the string variable first_name and the string literals "Hello, ", " (age ", and ")\n".

If you type in 22 Carlos, you'll see something that might be surprising until you think about it. The (misguided) input 22 Carlos will output

```
Hello, 22 (age -1)
```

The 22 will be read into first_name because, after all, 22 is a sequence of characters, and they are terminated by whitespace. On the other hand, Carlos isn't an integer, so it will not be read. The output will be 22 and age's initial value -1. Why? You didn't succeed in reading a value into it, so it kept its initial value.

A string read using >> is (by default) terminated by whitespace; that is, it reads a single word. But sometimes, we want to read more than one word. There are of course many ways of doing this. For example, we can read a name consisting of two words like this: AA

```
int main()
{
        cout << "Please enter your first and second names\n";
        string first;
        string second;
        cin >> first >> second;             // read two strings
        cout << "Hello, " << first << " " << second << '\n';
}
```

We simply used >> twice, once for each name. When we want to write the names to output, we must insert a space between them.

Note the absence of initializers for the two **strings** used as targets for input (**first** and **second**). By default, a **string** is initialized to the empty string, that is **""**.

> TRY THIS
>
> Get the "name and age" example to run. Then, modify it to write out the age in number of months: read the input in years and multiply (using the * operator) by 12. Read the age into a **double** to allow for children who can be very proud of being five and a half years old rather than just five.

2.4 Operations and operators

In addition to specifying what values can be stored in a variable, the type of a variable determines what operations we can apply to it and what they mean. For example:

```
int age = -1;
cin >> age;                      // >> reads an integer into age

string name;
cin >> name;                     // >> reads a string into name

int a2 = age+2;                  // + adds integers
string n2 = name + " Jr. ";      // + concatenates strings

int a3 = age-2;                  // - subtracts integers
string n3 = name - " Jr. ";      // error: - isn't defined for strings
```

XX By "error" we mean that the compiler will reject a program trying to subtract strings. The compiler knows exactly which operations can be applied to each variable and can therefore prevent many mistakes. However, the compiler doesn't know which operations make sense to you for which values, so it will happily accept legal operations that yield results that may look absurd to you. For example:

```
age = -100;
```

It may be obvious to you that you can't have a negative age (why not?) but nobody told the compiler, so it'll produce code for that definition.

Here is a table of useful operators for some common and useful types:

operation	bool	char	int	double	string
assignment	=	=	=	=	=
addition			+	+	
concatenation					+
subtraction			-	-	
multiplication			*	*	
division			/	/	
remainder (modulo)			%		

operation	bool	char	int	double	string
increment by 1			++	++	
decrement by 1			--	--	
increment by n			+= n	+= n	
add to end					+=
decrement by n			-= n	-= n	
multiply and assign			*=	*=	
divide and assign			/=	/=	
remainder and assign			%=		
read from s into x	s >> x	s >> x	s >> x	s >> x	s >> x
write x to s	s << x	s << x	s << x	s << x	s << x
equals	==	==	==	==	==
not equal	!=	!=	!=	!=	!=
greater than	>	>	>	>	>
greater than or equal	>=	>=	>=	>=	>=
less than	<	<	<	<	<
less than or equal	<=	<=	<=	<=	<=

A blank square indicates that an operation is not directly available for a type (though there may be indirect ways of using that operation; see §2.9).

We'll explain these operations, and more, as we go along. The key points here are that there are a lot of useful operators and that their meaning tends to be the same for similar types.

Let's try an example involving floating-point numbers:

```
int main()        // simple program to exercise operators
{
        cout << "Please enter a floating-point value: ";
        double n = 0;
        cin >> n;
        cout << "n == " << n
                << "\nn+1 == " << n+1
                << "\nthree times n == " << 3*n
                << "\ntwice n == " << n+n
                << "\nn squared == " << n*n
                << "\nhalf of n == " << n/2
                << "\nsquare root of n == " << sqrt(n)
                << '\n';
}
```

Obviously, the usual arithmetic operations have their usual notation and meaning as we know them from primary school. The exception is that the notation for equal is ==, rather than just =. Plain = is used for assignment. Naturally, not everything we might want to do to a floating-point number, such as taking its square root, is available as an operator. Many operations are represented as named functions. In this case, we use sqrt() from the standard library to get the square root of n: sqrt(n). The notation is familiar from math. We'll use functions along the way and discuss them in some detail in §3.5 and §7.4.

> TRY THIS
>
> Get this little program to run. Then, modify it to read an int rather than a double. Also, "exercise" some other operations, such as the modulo operator, %. Note that for ints / is integer division and % is remainder (modulo), so that 5/2 is 2 (and not 2.5 or 3) and 5%2 is 1. The definitions of integer *, /, and % guarantee that for two positive ints a and b we have a/b * b + a%b == a.

Strings have fewer operators, but they have plenty of named operations (PPP2.Ch23). However, the operators they do have can be used conventionally. For example:

```
int main()        // read first and second name
{
        cout << "Please enter your first and second names\n";
        string first;
        string second;
        cin >> first >> second;                    // read two strings

        string name = first + ' ' + second;        // concatenate strings
        cout << "Hello, " << name << '\n';
}
```

For strings, + means concatenation; that is, when s1 and s2 are strings, s1+s2 is a string where the characters from s1 are followed by the characters from s2. For example, if s1 has the value "Hello" and s2 the value "World", then s1+s2 will have the value "HelloWorld". Comparison of strings is particularly useful:

```
int main()        // read and compare names
{
        cout << "Please enter two names\n";
        string first;
        string second;
        cin >> first >> second;     // read two strings

        if (first == second)
                cout << "that's the same name twice\n";
        if (first < second)
                cout << first << " is alphabetically before " << second <<'\n';
        if (first > second)
                cout << first << " is alphabetically after " << second <<'\n';
}
```

Here, we used an if-statement, which will be explained in detail in §3.4.1.1, to select actions based on conditions.

2.5 Assignment and initialization

CC

In many ways, the most interesting operator is assignment, represented as =. It gives a variable a new value. For example:

```
int a = 3; // a starts out with the value 3
                                                        a: [ 3 ]
a = 4;          // a gets the value 3 (becomes 4)
                                                        a: [ 4 ]
int b = a;      // b starts out with a copy of a's value (that is, 4)
                                                        a: [ 4 ]
                                                        b: [ 4 ]
b = a+5;        // b gets the value a+5 (that is, 9)
                                                        a: [ 4 ]
                                                        b: [ 9 ]
a = a+7;        // a gets the value a+7 (that is, 11)
                                                        a: [ 11 ]
                                                        b: [ 9 ]
```

That last assignment deserves notice. First of all it clearly shows that `=` does not mean equals – clearly, `a` doesn't equal `a+7`. It means assignment, that is, to place a new value in a variable. What is done for `a=a+7` is the following: **XX**

[1] First, get the value of `a`; that's the integer 4.
[2] Next, add 7 to that 4, yielding the integer 11.
[3] Finally, put that 11 into `a`.

We can also illustrate assignments using strings:

```
sting a = "alpha";      // a starts out with the value "alpha"
                                                        a: [ alpha ]
a = "beta";             // a gets the value "beta" (becomes "beta")
                                                        a: [ beta ]
string b = a;           // b starts out with a copy of a's value (that is, "beta")
                                                        a: [ beta ]
                                                        b: [ beta ]
b = a+"gamma";          // b gets the value a+"gamma" (that is, "betagamma")
                                                        a: [ beta ]
                                                        b: [ betagamma ]
a = a+"delta";          // a gets the value a+"delta" (that is, "betadelta")
                                                        a: [ betadelta ]
                                                        b: [ betagamma ]
```

We use "starts out with" and "gets" to distinguish two similar, but logically distinct, operations: **CC**

- *Initialization*: giving a variable its initial value.
- *Assignment*: giving a variable a new value.

Logically assignment and initialization are different. In principle, an initialization always finds the variable empty. On the other hand, an assignment (in principle) must clear out the old value from the variable before putting in the new value. You can think of the variable as a kind of small box and the value as a concrete thing, such as a coin, that you put into it. Before initialization, the box is empty, but after initialization it always holds a coin so that to put a new coin in, you (i.e., the assignment operator) first have to remove the old one (''destroy the old value''). Things are not quite this literal in the computer's memory, but it's not a bad way of thinking of what's going on.

2.5.1 An example: detect repeated words

Assignment is needed when we want to put a new value into an object. When you think of it, it is obvious that assignment is most useful when you do things many times. We need an assignment when we want to do something again with a different value. Let's have a look at a little program that detects adjacent repeated words in a sequence of words. Such code is part of most grammar checkers:

```
int main()
{
        string previous;                        // previous word; initialized to ""
        string current;                         // current word
        while (cin>>current) {                  // read a stream of words
                if (previous == current)        // check if the word is the same as last
                        cout << "repeated word: " << current << '\n';
                previous = current;
        }
}
```

This program is not the most helpful since it doesn't tell where the repeated word occurred in the text, but it'll do for now. We will look at this program line by line starting with

```
string current;         // current word
```

By default, a `string` is initialized to the empty string, so we don't have to explicitly initialize it. We read a word into `current` using

```
while (cin>>current)
```

AA

This construct, called a `while`-statement, is interesting in its own right, and we'll examine it further in §3.4.2.1. The `while` says that the statement after `(cin>>current)` is to be repeated as long as the input operation `cin>>current` succeeds, and `cin>>current` will succeed as long as there are characters to read on the standard input. Remember that for a `string`, `>>` reads whitespace-separated words. You terminate this loop by giving the program an end-of-input character (usually referred to as *end of file*). On a Windows machine, that's Ctrl+Z (Control and Z pressed together) followed by an Enter (return). On a Linux machine, that's Ctrl+D (Control and D pressed together).

So, what we do is to read a word into `current` and then compare it to the previous word (stored in `previous`). If they are the same, we say so:

```
if (previous == current)        // check if the word is the same as last
    cout << "repeated word: " << current << '\n';
```

Then we have to get ready to do this again for the next word. We do that by copying the **current** word into **previous**:

```
previous = current;
```

This handles all cases provided that we can get started. What should we do for the first word where we have no previous word to compare? This problem is dealt with by the definition of **previous**:

```
string previous;        // previous word; initialized to ""
```

The empty string is not a word. Therefore, the first time through the **while**-statement, the test

```
if (previous == current)
```

fails (as we want it to).

One way of understanding program flow is to "play computer," that is, to follow the program line for line, doing what it specifies. Just draw boxes on a piece of paper and write their values into them. Change the values stored as specified by the program.

AA

TRY THIS

Execute this program yourself using a piece of paper. Use the input **The cat cat jumped**. Even experienced programmers use this technique to visualize the actions of small sections of code that somehow don't seem completely obvious.

TRY THIS

Get the "repeated word detection program" to run. Test it with the sentence **She she laughed "he he he!" because what he did did not look very very good good**. How many repeated words were there? Why? What is the definition of *word* used here? What is the definition of *repeated word*? (For example, is **She she** a repetition?)

2.5.2 Composite assignment operators

Incrementing a variable (that is, adding 1 to it) is so common in programs that C++ provides a special syntax for it. For example:

```
++counter
```

means

```
counter = counter + 1
```

There are many other common ways of changing the value of a variable based on its current value. For example, we might like to add 7 to it, to subtract 9, or to multiply it by 2. Such operations are also supported directly by C++. For example:

```
a += 7;    // means a = a+7
b -= 9;    // means b = b-9
c *= 2;    // means c = c*2
```

In general, for any binary operator **oper**, **a oper= b** means **a = a oper b**. For starters, that rule gives us operators **+=**, **−=**, ***=**, **/=**, and **%=**. This provides a pleasantly compact notation that directly reflects our ideas. For example, in many application domains ***=** and **/=** are referred to as "scaling."

2.5.3 An example: find repeated words

Consider the example of detecting repeated adjacent words above. We could improve that by giving an idea of where the repeated word was in the sequence. A simple variation of that idea simply counts the words and outputs the count for the repeated word:

```
int main()
{
	int number_of_words = 0;
	string previous;			// previous word; initialized to ""
	string current;
	while (cin>>current) {
		++number_of_words;		// increase word count
		if (previous == current)
			cout << "word number " << number_of_words << " repeated: " << current << '\n';
		previous = current;
	}
}
```

We start our word counter at **0**. Each time we see a word, we increment that counter:

```
++number_of_words;
```

That way, the first word becomes number **1**, the next number **2**, and so on. We could have accomplished the same by saying

```
number_of_words += 1;
```

or even

```
number_of_words = number_of_words+1;
```

but **++number_of_words** is shorter and expresses the idea of incrementing directly.

AA

Note how similar this program is to the one from §2.5.1. Obviously, we just took the program from §2.5.1 and modified it a bit to serve our new purpose. That's a very common technique: when we need to solve a problem, we look for a similar problem and use our solution for that with suitable modification. Don't start from scratch unless you really have to. Using a previous version of a program as a base for modification often saves a lot of time, and we benefit from much of the effort that went into the original program.

2.6 Names

We name our variables so that we can remember them and refer to them from other parts of a program. What can be a name in C++? In a C++ program, a name starts with a letter and contains only letters, digits, and underscores. For example:

```
x
number_of_elements
Fourier_transform
z2
Polygon
```

The following are not names:

```
2x               // a name must start with a letter
time@to@market   // @ is not a letter, digit, or underscore
Start menu       // space is not a letter, digit, or underscore
```

When we say "not names," we mean that a C++ compiler will not accept them as names.

If you read system code or machine-generated code, you might see names starting with underscores, such as _foo. Never write those yourself; such names are reserved for implementation and system entities. By avoiding leading underscores, you will never find your names clashing with some name that the implementation generated. **XX**

Names are case sensitive; that is, uppercase and lowercase letters are distinct, so x and X are different names. This little program has at least four errors:

```
import std;

int Main()
{
	STRING s = "Goodbye, cruel world! ";
	cOut << S << '\n';
}
```

It is usually not a good idea to define names that differ only in the case of a character, such as one and One; that will not confuse a compiler, but it can easily confuse a programmer.

> TRY THIS
>
> Compile the "Goodbye, cruel world!" program and examine the error messages. Did the compiler find all the errors? What did it suggest as the problems? Did the compiler get confused and diagnose more than four errors? Remove the errors one by one, starting with the lexically first, and see how the error messages change (and improve).

The C++ language reserves many names as *keywords*, such as if, else, class, int, and module. You can find complete lists on the Web (e.g., cppreference.com). You can't use those to name your variables, types, functions, etc. For example: **XX**

```
int if = 7;    // error: if is a keyword
```

You can use names of facilities in the standard library, such as string for your own variables, but you shouldn't. Reuse of such a common name will cause trouble if you should ever want to use the standard library:

```
int string = 7;  // this will lead to trouble
```

AA

When you choose names for your variables, functions, types, etc., choose meaningful names; that is, choose names that will help people understand your program. Even you will have problems understanding what your program is supposed to do if you have littered it with variables with ‘‘easy to type’’ names like **x1**, **x2**, **s3**, and **p7**. Abbreviations and acronyms can confuse people, so use them sparingly. These acronyms were obvious to us when we wrote them, but we expect you’ll have trouble with at least one:

```
mtbf    TLA    myw    NBV
```

We expect that in a few months, we’ll also have trouble with at least one.

Short names, such as **x** and **i**, are meaningful when used conventionally; that is, **x** should be a local variable or a parameter (see §3.5 and §7.4) and **i** should be a loop index (§3.4.2.3).

Don’t use overly long names; they are hard to type, make lines so long that they don’t fit on a screen, and are hard to read quickly. These are probably OK:

```
partial_sum    element_count    stable_partition
```

These are probably too long:

```
the_number_of_elements    remaining_free_slots_in_symbol_table
```

Our ‘‘house style’’ is to use underscores to separate words in an identifier, such as **element_count**, rather than alternatives, such as **elementCount** and **ElementCount**. We never use names with all capital letters, such as **ALL_CAPITAL_LETTERS**, because that’s conventionally reserved for macros (PPP2.§27.8), which we avoid. We use an initial capital letter for types we define, such as **Square** and **Graph**. The C++ language and standard library don’t use the initial-capital-letter style, so it’s **int** rather than **Int** and **string** rather than **String**. Thus, our convention helps to minimize confusion between our types and the standard ones.

XX

Avoid names that are easy to mistype, misread, or confuse. For example:

```
Name    names    nameS    foo    f00    fl    f1    fI    fi
```

The characters **0** (zero), **o** (lowercase **O**), **O** (uppercase **o**), **1** (one), **I** (uppercase **i**), and **l** (lowercase **L**) are particularly prone to cause trouble.

2.7 Types and objects

CC

The notion of type is central to C++ and most other programming languages. Let’s take a closer and slightly more technical look at types:

- A *type* defines a set of possible values and a set of operations (for an object).
- An *object* is some memory that holds a value of a given type.
- A *value* is a set of bits in memory interpreted according to a type.
- A *variable* is a named object.
- A *declaration* is a statement that gives a name and a type to an object.
- A *definition* is a declaration that sets aside memory for an object.

Informally, we think of an object as a box into which we can put values of a given type. An **int** box can hold integers, such as **7**, **42**, and **–399**. A **string** box can hold character string values, such as

"Interoperability", "operators: +−*/%", and "Old MacDonald had a farm". Graphically, we can think of it like this:

```
int a = 7;                      a:  | 7 |
int b = 9;                      b:  | 9 |
char c = 'a';                   c:  | a |
double x = 1.2;                 x:  | 1.2 |
string s1 = "Hello, World!";    s1: | 13 | Hello, World! |
string s2 = "1.2";              s2: | 3 | 1.2 |
```

The representation of a string is a bit more complicated than that of an int because a string keeps track of the number of characters it holds. Note that a double stores a number whereas a string stores characters. For example, x stores the number 1.2, whereas s2 stores the three characters '1', '.', and '2'. The quotes for character and string literals are not stored.

Every int is of the same size; that is, the compiler sets aside the same fixed amount of memory for each int. On a typical computer or phone, that amount is 4 bytes (32 bits). Similarly, bools, chars, and doubles are fixed size. You'll typically find that a computer uses a byte (8 bits) for a bool or a char and 8 bytes for a double. Note that different types of objects take up different amounts of space. In particular, a char takes up less space than an int, and string differs from double, int, and char in that different strings can take up different amounts of space.

CC

The meaning of bits in memory is completely dependent on the type used to access it. Think of it this way: computer memory doesn't know about our types; it's just memory. The bits of memory get meaning only when we decide how that memory is to be interpreted. This is similar to what we do every day when we use numbers. What does 12.5 mean? We don't know. It could be $12.5, 12.5cm, or 12.5 gallons. Only when we supply the unit does the notation 12.5 mean anything.

For example, the very same bits of memory that represent the integer value 120 when looked upon as an int would be the character 'x' when looked upon as a char. If looked at as a string, it wouldn't make sense at all and would become a run-time error if we tried to use it. We can illustrate this graphically like this, using 1 and 0 to indicate the value of bits in memory:

```
00000000 00000000 00000000 01111000
```

This is the setting of the bits of an area of memory (a word) that could be read as an int (120) or as a char ('x', looking at the rightmost 8 bits only). A *bit* is a unit of computer memory that can hold the value 0 or 1.

2.8 Type safety

AA

Every object is given a type when it is defined, and that type never changes. A program – or a part of a program – is type-safe when all objects are used only according to the rules for their type. Complete type safety is the ideal and the general rule for the language. Unfortunately, a C++

compiler cannot by itself guarantee complete type safety for arbitrary code, so we must avoid unsafe techniques. That is, we must obey some coding rules to achieve type safety. There are ways of enforcing such rules, but historically such rules have been considered overly restrictive and have not been consistently enforced. Given older versions of C++ (earlier than recent ISO C++ standards) and techniques adopted from the C language, this was unavoidable and therefore not unreasonable. However, when using modern C++ and modern analysis tools, type safety can be verified for most uses of C++. The ideal is never to use language features that cannot be proven type-safe before the program starts executing: *static type safety*. With the obvious exception of code used to illustrate unsafe techniques (e.g., §16.1.1), the code in this book follows the C++ Core Guidelines [CG] and has been verified to be type safe.

XX The ideal of type safety is incredibly important when writing reliable code. That's why we spend time on it this early in the book. Please note the pitfalls and avoid them. If you don't, you will face much frustration and your code will contain many obscure errors.

AA For example, using an uninitialized variable is not type-safe:

```
int main()
{
        double x;                   // we "forgot" to initialize: the value of x is undefined
        double y = x;               // the value of y is undefined
        double z = 2.0+x;           // the meaning of + and the value of z are undefined
}
```

AA Always initialize your variables! Implementations can easily enforce this rule, but unfortunately they typically don't do so by default. Except, fortunately, for types such as **string** and **vector** where default initialization is guaranteed (§2.5, §7.2.3, §8.4.2). Figure out how to enable warnings (often a **–Wall** compiler option) and adhere to them. Doing so will save you a lot of grief.

CC Modern C++ implementations also come with significant static-analysis tools that allow us to prevent more subtle problems. Professionals use such tools extensively, and so will you if you become or aim to become a professional, but at this initial stage of learning just follow the rules and styles used in this book.

2.9 Conversions

In §2.4, we saw that we can't directly add **char**s or compare a **double** to an **int**. However, C++ provides indirect ways to do both. When needed in an expression, a **char** is converted to an **int** and an **int** is converted to a **double**. For example:

```
char c = 'x';
int i1 = c;                 // i1 gets the integer value of c
int i2 = c+1000;            // i2 gets the integer value of c added to 1000
double d = i2+7.3;          // d gets the floating-point value of i2 plus 7.3
```

Here, **i1** gets the value **120**, which is the integer value of the character **'x'** in the popular 8-bit character set, ASCII. This is a simple way of getting the numeric representation of a character. To get the value of **i2**, the addition is done using integer arithmetic and gives the value **1120**. The **char** is said to be *promoted* to **int** before the addition.

Similarly, when having a mixture of floating-point values and integer values, the integers are promoted to floating point to give unsurprising results. Here, **d** gets the value **1127.3**.

Conversions are of two kinds

- *widening*: Conversions that preserve information, such as **char** to **int**.
- *narrowing*: Conversions that may lose information, such as **int** to **char**.

A widening conversion converts a value to an equal value or to the best approximation of an equal value. Widening conversions are usually a boon to the programmer and simplify writing code.

Unfortunately, C++ also allows for implicit narrowing conversions. By narrowing, we mean that a value is turned into a value of another type that does not equal the original value. XX

Consider **int** and **char**. Conversions from **char** to **int** don't have problems with narrowing. However, a **char** can hold only very small integer values. Often, a **char** is an 8-bit byte whereas an **int** is 4 bytes:

We can't put a large number, such as **1000**, into a **char**. Such conversions are called *narrowing* because they put a value into an object that may be too small ("narrow") to hold all of it. Unfortunately, conversions such as **double** to **int** and **int** to **char** are by default accepted by most compilers even though they are narrowing. Why can this be a problem? Because often we don't suspect that a narrowing – information destroying – conversion is taking place. Consider: XX

```
double x = 2.7;
// ... lots of code ...
int y = x;          // y becomes 2
```

By the time we assign **x** to **y** we may have forgotten that **x** was a **double**, or that a **double**-to-**int** conversion *truncates* (always rounds down, toward zero) rather than using the conventional 4/5 rounding (rounding towards the nearest integer). What happens is well-defined, but there is nothing in the **y = x;** to remind us that information (the **.7**) is thrown away.

To get a feel for conversions and an understanding why narrowing conversions must be avoided, experiment. Consider this program that shows how conversions from **double** to **int** and conversions from **int** to **char** are done on your machine:

```
int main()
{
     double d = 0;
     while (cin>>d) {          // repeat the statements below as long as we type in numbers
          int i = d;           // try to squeeze a floating-point value into an integer value
          char c = i;          // try to squeeze an integer into a char
          cout << "d==" << d              // the original double
               << " i=="<< i              // double converted to int
               << " c==" << c             // int value of char
               << " char(" << c << ")\n";    // the char
     }
}
```

> TRY THIS
>
> Run this program with a variety of inputs:
>
> - Small values (e.g., 2 and 3).
> - Large values (larger than 127, larger than 1000).
> - Negative values.
> - 56, 89, and 128.
> - Non-integer values (e.g., 56.9 and 56.2).
>
> You'll find that many inputs produce "unreasonable" results when converted. Basically, we are trying to put a gallon into a pint pot (about 4 liters into a 500ml glass).

CC

Why do people accept the problem of narrowing conversions? The major reason is history: C++ inherited narrowing conversions from its ancestor language, C, so from day one of C++, there existed much code that depended on narrowing conversions. Also, many such conversions don't actually cause problems because the values involved happen to be in range, and many programmers object to "compilers telling them what to do." In particular, the problems with narrowing conversions are often manageable in small programs and for experienced programmers. They can be a source of errors in larger programs, though, and a significant cause of problems for novice programmers. Fortunately, compilers can warn about narrowing conversions – and many do. Adhere to those warnings.

When we really need narrowing, we can use **narrow<T>(x)** to check that **x** can be narrowed to a **T** without loss of information (§7.4.7). When we want rounding, we can use **round_to<int>(x)**. Both are supplied by **PPP_support**.

CC

For historical and practical reasons, C++ offers four notations for initialization: For example:

```
int x0 = 7.8;          // narrows, some compilers warn
int x1 {7.8};          // error: {} doesn't narrow
int x2 = {7.8};        // error: ={} doesn't narrow (the redundant = is allowed)
int x3 (7.8);          // narrows, some compilers warn
```

The **=** and **={}** notations go back to the early days of C. We use the **=** notation when an initialization simply copies its initializer and the **{}** and **={}** notations for more complex initializations and when we want compile-time protection against narrowing.

```
int x = 7;
double d = 7.7;
string s = "Hello, World\n";

vector v = {1, 2, 3, 5, 8 };        // see §17.3
pair p {"Hello",17};                // see §20.2.2
```

We reserve **()** initialization to a few very special cases (§17.3).

2.10 Type deduction: auto

You may have noticed a bit of repetitiveness in definitions. Consider:

```
int x = 7;
double d = 7.7;
```

We know that 7 is an integer and that 7.7 is a floating-point number, and so does the compiler. Why then do we have to say int and double? Well, we don't have to unless we want to; we can let the compiler deduce the type from the type of the initializer:

```
auto x = 7;          // x is an int (because 7 is)
auto d = 7.7;        // d is a double (because 7.7 is)
```

AA

This version using auto means *exactly* the same as the one with explicit types. We use auto when, and *only* when, the type is obvious from the initializer and we don't want any conversion.

When we use longer type names (§18.5.2, §20.2.1) and in generic programming (§18.1.2) the notational convenience of auto becomes significant. For example:

```
auto z = complex<double>{1.3,3.4};
auto p = make_unique<Pair<string,int>>{"Harlem",10027};    // a unique_ptr<Pair<string,int>> (§18.5.2)
auto b = lst.begin();                                      // lst.begin is a vector<int>::iterator (§19.3.2)
```

Until then, resist the temptation to overuse auto. Overuse of auto can make code obscure so that we get unpleasant surprises.

Drill

After each step of this drill, run your program to make sure it is really doing what you expect it to. Keep a list of what mistakes you make so that you can try to avoid those in the future.

[1] Write a program that produces a simple form letter based on user input. Begin by typing the code from §2.1 prompting a user to enter his or her first name and writing "Hello, first_name" where first_name is the name entered by the user. Then modify your code as follows: change the prompt to "Enter the name of the person you want to write to" and change the output to "Dear first_name,". Don't forget the comma.

[2] Add an introductory line or two, like "How are you? I am fine. I miss you." Be sure to indent the first line. Add a few more lines of your choosing – it's your letter.

[3] Now prompt the user for the name of another friend and store it in friend_name. Add a line to your letter: "Have you seen friend_name lately?"

[4] Prompt the user to enter the age of the recipient and assign it to an int variable age. Have your program write "I hear you just had a birthday and you are age years old." If age is 0 or less or 110 or more, call simple_error("you're kidding!") using simple_error() from PPP_support.

[5] Add this to your letter:

> If your friend is under 12, write "Next year you will be age+1." If your friend is 17, write "Next year you will be able to vote." If your friend is over 70, write "Are you retired?"

Check your program to make sure it responds appropriately to each kind of value.

[6] Add "Yours sincerely," followed by two blank lines for a signature, followed by your name.

Review

[1] What is meant by the term *prompt*?
[2] Which operator do you use to read into a variable?
[3] What notations can you use to initialize an object?
[4] If you want the user to input an integer value into your program for a variable named **number**, what are two lines of code you could write to ask the user to do it and to input the value into your program?
[5] What is **\n** called and what purpose does it serve?
[6] What terminates input into a string?
[7] What terminates input into an integer?
[8] How would you write the following as a single line of code:

```
cout << "Hello, ";
cout << first_name;
cout << "!\n";
```

[9] What is an object?
[10] What is a literal?
[11] What kinds of literals are there?
[12] What is a variable?
[13] What are typical sizes for a **char**, an **int**, and a **double**?
[14] What measures do we use for the size of small entities in memory, such as **int**s and **string**s?
[15] What is the difference between **=** and **==**?
[16] What is a definition?
[17] What is an initialization and how does it differ from an assignment?
[18] What is string concatenation and how do you make it work in C++?
[19] What operators can you apply to an **int**?
[20] Which of the following are legal names in C++? If a name is not legal, why not?

This_little_pig	This_1_is fine	2_For_1_special	latest thing
George@home	_this_is_ok	MineMineMine	number
correct?	stroustrup.com	$PATH	

[21] Give five examples of legal names that you shouldn't use because they are likely to cause confusion.
[22] What are some good rules for choosing names?
[23] What is type safety and why is it important?
[24] Why can conversion from **double** to **int** be a bad thing?
[25] Define a rule to help decide if a conversion from one type to another is safe or unsafe.
[26] How can we avoid undesirable conversions?
[27] What are the uses of **auto**?

Terms

assignment	definition	operation	cin
increment	operator	concatenation	initialization
type	conversion	name	type safety
declaration	narrowing	value	decrement
object	variable	widening	truncation
int	double	string	auto
==	!=	=	++
<	<=	>	>=

Exercises

[1] If you haven't done so already, do the TRY THIS exercises from this chapter.

[2] Write a program in C++ that converts from miles to kilometers. Your program should have a reasonable prompt for the user to enter a number of miles. Hint: A mile is1.609 kilometers.

[3] Write a program that doesn't do anything, but declares a number of variables with legal and illegal names (such as int double = 0;), so that you can see how the compiler reacts.

[4] Write a program that prompts the user to enter two integer values. Store these values in int variables named val1 and val2. Write your program to determine the smaller, larger, sum, difference, product, and ratio of these values and report them to the user.

[5] Modify the program above to ask the user to enter floating-point values and store them in double variables. Compare the outputs of the two programs for some inputs of your choice. Are the results the same? Should they be? What's the difference?

[6] Write a program that prompts the user to enter three integer values, and then outputs the values in numerical sequence separated by commas. So, if the user enters the values 10 4 6, the output should be 4, 6, 10. If two values are the same, they should just be ordered together. So, the input 4 5 4 should give 4, 4, 5.

[7] Do exercise 6, but with three string values. So, if the user enters the values Steinbeck, Hemingway, Fitzgerald, the output should be Fitzgerald, Hemingway, Steinbeck.

[8] Write a program to test an integer value to determine if it is odd or even. As always, make sure your output is clear and complete. In other words, don't just output yes or no. Your output should stand alone, like The value 4 is an even number. Hint: See the remainder (modulo) operator in §2.4.

[9] Write a program that converts spelled-out numbers such as "zero" and "two" into digits, such as 0 and 2. When the user inputs a number, the program should print out the corresponding digit. Do it for the values 0, 1, 2, 3, and 4 and write out not a number I know if the user enters something that doesn't correspond, such as stupid computer!.

[10] Write a program that takes an operation followed by two operands and outputs the result. For example:

```
+ 100 3.14
* 4 5
```

Read the operation into a string called **operation** and use an **if**-statement to figure out which operation the user wants, for example, **if (operation=="+")**. Read the operands into variables of type **double**. Implement this for operations called **+**, **–**, *, **/**, **plus**, **minus**, **mul**, and **div** with their obvious meanings.

[11] Write a program that prompts the user to enter some number of pennies (1-cent coins), nickels (5-cent coins), dimes (10-cent coins), quarters (25-cent coins), half dollars (50-cent coins), and one-dollar coins (100-cent coins). Query the user separately for the number of each size coin, e.g., ‘‘How many pennies do you have?’’ Then your program should print out something like this:

```
You have 23 pennies.
You have 17 nickels.
You have 14 dimes.
You have 7 quarters.
You have 3 half dollars.
The value of all of your coins is 573 cents.
```

Make some improvements: if only one of a coin is reported, make the output grammatically correct, e.g., **14 dimes** and **1 dime** (not **1 dimes**). Also, report the sum in dollars and cents, i.e., **.73** instead of **573 cents**.

Postscript

Please don’t underestimate the importance of the notion of type safety. Types are at the center of most notions of correct programs, and some of the most effective techniques for constructing programs rely on the design and use of types – as you’ll see in Chapter 5 and Chapter 8, and indeed in most of the rest of the book.

3

Computation

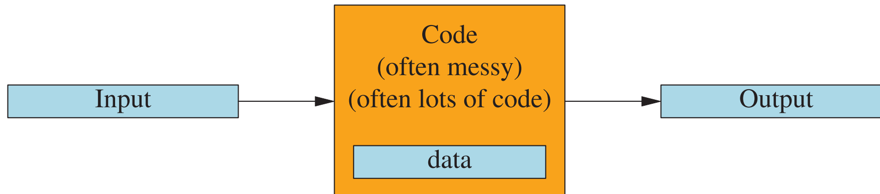
If it doesn´t have
to produce correct results,
I can make it arbitrarily fast.
– Gerald M. Weinberg

This chapter presents the basics of computation. In particular, we discuss how to compute a value from a set of operands (*expression*), how to choose among alternative actions (*selection*), and how to repeat a computation for a series of values (*iteration*). We also show how a particular sub-computation can be named and specified separately (a *function*). Our primary concern is to express computations in ways that lead to correct and well-organized programs. To help you perform more realistic computations, we introduce the **vector** type to hold sequences of values.

§3.1 Computation
§3.2 Objectives and tools
§3.3 Expressions
Constant expressions; Operators
§3.4 Statements
Selection; Iteration
§3.5 Functions
Why bother with functions?; Function declarations
§3.6 **vector**
Traversing a **vector**; Growing a **vector**; A numeric example; A text example
§3.7 Language features

3.1 Computation

From one point of view, all that a program ever does is to compute; that is, it takes some inputs and produces some output. After all, we call the hardware on which the program runs "a computer." This view is reasonable as long as we take a broad view of what constitutes input and output:

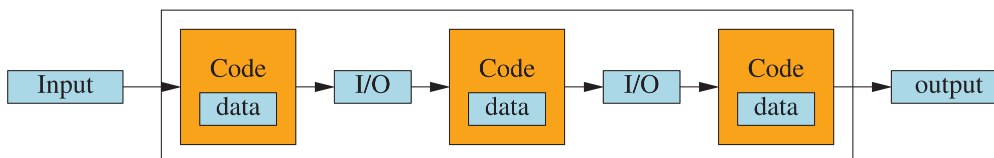


The input can come from a keyboard, from a mouse, from a touch screen, from files, from other input devices, from other programs, from other parts of a program. "Other input devices" is a category that contains really interesting input sources: music keyboards, video recorders, network connections, temperature sensors, image sensors, power supplies, etc.

To deal with input, a program usually contains some data, sometimes referred to as its *data structures* or its *state*. For example, a calendar program may contain lists of holidays in various countries and a list of your appointments. Some of that data is part of the program from the start; other data is built up as the program reads input and collects useful information from it. For example, the calendar program will probably build your list of appointments from the input you give it. For the calendar, the main inputs are the requests to see the months and days you ask for (probably using mouse clicks) and the appointments you give it to keep track of (probably by typing information on your keyboard). The output is the display of calendars and appointments, plus the buttons and prompts for input that the calendar program writes on your screen. In addition, the calendar may send you reminders and synchronize with other copies of the calendar programs.

Input comes from a wide variety of sources. Similarly, output can go to a wide variety of destinations. Output can be to a screen, to files, to network connections, to other output devices, to other programs, and to other parts of a program. Examples of output devices include network interfaces, music synthesizers, electric motors, light generators, heaters, etc.

From a programming point of view the most important and interesting categories are "to/from another program" and "to/from other parts of a program." Most of the rest of this book could be seen as discussing that last category: how do we express a program as a set of cooperating parts and how can they share and exchange data? These are key questions in programming. We can illustrate that graphically:



The abbreviation *I/O* stands for "input/output." In this case, the output from one part of code is the input for the next part. What such "parts of a program" share is data stored in main memory, on persistent storage devices (such as disks), or transmitted over network connections. By "parts of a program" we mean entities such as a function producing a result from a set of input arguments (e.g., a square root from a floating-point number), a function performing an action on a physical object (e.g., a function drawing a line on a screen), or a function modifying some table within the program (e.g., a function adding a name to a table of customers).

When we say "input" and "output" we generally mean information coming into and out of a computer, but as you see, we can also use the terms for information given to or produced by a part of a program. Inputs to a part of a program are often called *arguments* and outputs from a part of a program are often called *results*.

By *computation* we simply mean the act of producing some outputs based on some inputs, such as producing the result (output) **49** from the argument (input) **7** using the computation (function) **square** (see §3.5). As a possibly helpful curiosity, we note that until the 1950s a computer was defined as a person who did computations, such as an accountant, a navigator, or a physicist. Today, we simply delegate most computations to computers of various forms, such as smartphones.

3.2 Objectives and tools

CC

Our job as programmers is to express computations

- Correctly
- Simply
- Efficiently

Please note the order of those ideals: it doesn't matter how fast a program is if it gives the wrong results. Similarly, a correct and efficient program can be so complicated that it must be thrown away or completely rewritten to produce a new version (release). Remember, useful programs will always be modified to accommodate new needs, new hardware, etc. Therefore a program – and any part of a program – should be as simple as possible to perform its task. For example, assume that you have written the perfect program for teaching basic arithmetic to children in your local school, and that its internal structure is a mess. Which language did you use to communicate with the children? English? English and Spanish? What if I'd like to use it in Finland? In Kuwait? How would you change the (natural) language used for communication with a child? If the internal structure of the program is a mess, the logically simple (but in practice almost always very difficult) operation of changing the natural language used to communicate with users becomes insurmountable.

AA

Concerns about correctness, simplicity, and efficiency become ours the minute we start writing code for others and accept the responsibility to do that well; that is, we must accept that responsibility when we decide to become professionals. In practical terms, this means that we can't just throw code together until it appears to work; we must concern ourselves with the structure of code. Paradoxically, concerns for structure and "quality of code" are often the fastest ways of getting something to work. When programming is done well, such concerns minimize the need for the most frustrating part of programming: debugging; that is, good program structure during development can minimize the number of mistakes made and the time needed to search for such errors and to remove them.