

O'REILLY®

Introducing C++

The Easy Way to
Start Learning
Modern C++



Frances Buontempo
Foreword by Kevlin Henney

Praise for *Introducing C++*

It's wonderful to see a fresh book that not only uses C++ to teach programming from scratch but that starts with modern C++23! With the number of C++ programmers worldwide growing every year with no sign of slowing down, this book is very timely.

—Herb Sutter, ISO C++ committee chair

This is the book I wish I'd had under my pillow when I started with C++ many years ago after coming from a long career using other programming languages.

—Daniela Engert, senior software engineer at GMH Prüftechnik GmbH and member of the C++ standardization committee

Finally, a C++ book that saves the sharp edges for later and lets you build real things first. Modern, practical, and long overdue.

—Matt Godbolt, *Compiler Explorer*

I've been out of the C++ sphere for some time, but Buontempo's text is clear and the explanations are excellent. The chapter on Lambdas was particularly helpful for me as a total newbie--I didn't understand all the fuss about Lambdas until I read it.

—Emyr Williams, ACM member

Introducing C++

The Easy Way to Start Learning Modern C++

Frances Buontempo
Foreword by Kevlin Henney

O'REILLY®

Introducing C++

by Frances Buontempo

Copyright © 2026 Frances Buontempo. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Brian Guerin

Development Editor: Sarah Grey

Production Editor: Ashley Stussy

Copyeditor: Emily Wydeven

Proofreader: Kim Wimpsett

Indexer: nSight, Inc.

Cover Designer: Susan Brown

Cover Illustrator: José Marzan Jr.

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

March 2026: First Edition

Revision History for the First Edition

2026-03-10: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098178147> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Introducing C++*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17814-7

[LSI]

To the memory of our cat Vim



Table of Contents

Foreword.....	xiii
Preface.....	xv
1. Hello, World!.....	1
Installing Tools	3
Linux and macOS	3
Windows	4
Using Your Tools	4
Running Your Program	7
Writing to the Screen	7
Using println	8
Troubleshooting	10
Using cout	11
Understanding println and cout in Depth	12
Conclusion	14
2. Variables and Keyboard Input.....	15
Declaring Variables	15
Character Input	16
Detecting Input Problems	18
Input of Real Numbers	18
Detecting Problems with Numbers	19
Detecting More General Problems	22
A Function for Input with Some Tests	24
Starting with a Failing Test	24
Breaking Your Code into Functions	26
Starting with a Failing Test, Again	27

Testing Bad Input	29
Refactor	31
Calling Your New Function from main	32
Understanding Variables, std::cin, and Functions in Depth	33
Clearing Input Errors	34
More on Functions	37
Conclusion	41
3. Exceptions and Expectations.....	43
Exceptions	43
Throwing Exceptions	46
Trying and Catching	47
Handling Exceptions with a try/catch Block	47
Expectations	49
Understanding Exceptions and Expectations in More Depth	51
Other Exception Types	52
Position of catch Blocks	54
Expected Without a Value	55
Conclusion	56
4. Using Loops, Arrays, and Vectors.....	59
Input of Several Numbers Using a Loop	59
A while Loop	60
Using an Array	62
Displaying and Using the Numbers	68
Using a Vector	71
Adding More Elements to a Vector	72
A Few Other Container Functions	74
Getting Several Numbers in a Vector	75
Understanding Sequential Containers in More Depth	76
Initializing Containers with an Initializer List	76
What Happens When You Add to a Vector	77
What Happens When You Delete from a Vector	78
Initializing a Vector with a Fixed Value	79
Other Sequential Containers	80
Conclusion	80
5. Using Standard Library Algorithms.....	83
Getting Several Numbers Into a Vector (Again)	83
Analyzing Your Numbers Using Algorithms	89
Using Predicates in Algorithms	91
Using Iterators in Algorithms	94

The Old Way to Remove Items	95
Finding an Average with an Algorithm	99
Understanding Algorithms in More Depth	103
Using for Loops	103
Binary Operators and Predicates	106
More on Iterators	107
Conclusion	107
6. Lambdas and the Ranges Library.....	109
Removing Negative Numbers Using a Lambda	109
Using a Lambda to Vary Behavior via <code>std::function</code>	111
Filtering Out Negative Numbers Using the Ranges' View	117
Using Lambda Captures for Fun and Profit	120
Understanding Lambdas and Views in More Depth	123
Lambda Captures by Value	123
Lambda Captures by Reference	127
Composing Views	128
Lazy Views	132
Conclusion	133
7. Random Numbers.....	135
Generating Random Numbers	136
Writing an Overload for a Function	139
Building a Trading Game	143
Understanding Code with Random Numbers (and Vectors) in Depth	146
Using a Normal Distribution	147
Considerations for Code That Uses Random Numbers	150
Creating and Filling Vectors	154
Conclusion	156
8. Working with Files.....	159
Writing to a File	159
Detecting and Reporting Problems	161
Using the Filesystem Library	163
Reading from a File	165
Understanding Files in Depth	167
Different File Modes	167
Bitwise Operators and Bitmasks	169
Reading Previous Prices	171
Conclusion	175

9. Strings and Formatting.....	177
C-Style String Literals and Characters	177
Providing Arguments to main	179
Creating and Manipulating a <code>std::string</code>	180
Other Ways to Create a <code>std::string</code>	181
More <code>std::string</code> Functions	182
String Views	184
Formatting and More on <code>std::println</code>	187
<code>std::format</code> and Format Specifications	188
An Improved Trading Game	190
Understanding Strings in Depth	195
Joining <code>std::strings</code> Efficiently	195
Using <code>std::println</code> to Save to a File	196
Conclusion	198
10. Classes: Member Variables and Member Functions.....	201
A Simple Class	202
Private and Public Access Specifiers	205
Constructors and Destructors	208
Using the Stock Class in a <code>std::vector</code>	211
Introducing Classes in Depth	214
Constructors and Destructors in Depth	215
Splitting a Class Between Header and Source Files	217
Conclusion	219
11. Classes: Special Member Functions and Move Semantics.....	221
Copying Objects	221
Moving Objects	223
Move and Copy Assignments	225
Copies and Moves in Depth	229
How Does a <code>std::string</code> Work?	229
Move Constructors and Move Assignments	231
Copy Constructors and Copy Assignments	232
Conclusion	233
12. Memory Management with <code>std::unique_ptr</code>.....	235
Creating a <code>std::unique_ptr</code>	236
Using a <code>std::unique_ptr</code>	238
Smart Pointers in Depth	239
More on Pointers and References	239
Unique Pointers in More Detail	241
Other Smart Pointers	243

Custom Deleters	244
Using a <code>std::unique_ptr</code> in a Class	245
Using the Exchange Class	249
Conclusion	252
13. Classes: Virtual Functions and Inheritance.....	253
Base Class and Derived Classes	254
Defining an Abstract Base Class	254
A Derived Class	255
Using Derived Classes	258
A New Trading Game	261
Adding Another Derived Type	264
Virtual Functions and Inheritance in Depth	271
Virtual Destructors	271
Virtual Functions and Slicing	273
Conclusion	275
14. Using <code>std::variant</code> and <code>std::visit</code>.....	277
Creating and Using a <code>std::variant</code>	278
Using a <code>std::variant</code> in <code>std::visit</code>	281
Using the Event in the Trading Game	284
The <code>std::variant</code> in Depth	287
Spotting and Handling Potential Problems with <code>std::variant</code>	288
Using <code>std::optional</code> and <code>std::any</code>	289
Conclusion	291
15. Templates and <code>std::unordered_map</code>.....	293
Making a Lookup Table	293
Write Your Own Template	295
Specializing <code>std::hash</code>	299
Adding an Equality Operator for an Event	302
Adding a Way to Display the Events	304
Keeping a Tally of Events in Your Trading Game	306
Associative Containers and Templates in Depth	309
More on Templates	311
Conclusion	314
Index.....	315

Foreword

C++ is a big language, but it is not well served by a big book. This is especially true for an introduction to the language. An introduction is a beginning, not an everything. You want preparation, not intimidation. You want an on-ramp, not the north face of the Eiger. The first steps should encourage you to take the next steps rather than scare you off the path.

And that's the beginning Fran has given you. This introduction is complete, but not in the sense that it braindumps everything a C++ expert of many decades standing might know. It equips you with the skills to write real code, the freedom to choose where you want to go next, and the confidence to follow through. This book may be the beginning of your C++ journey, but it's also a journey in its own right, one that starts in the foothills of "Hello, world!" and travels into the world of stock prices and asset trading.

But make no mistake, this is not some passive tourist experience where you cruise past the sights of the language and its library from the window of a tour bus. This is a walking route that takes in landmarks and alleyways, paved streets and potholes, working code and compilation errors. You'll get a feel for the place and what it's like to live and work there. Fran is your guide. She knows the sights, the sites, and how to steer you clear of any trouble.

This is a book of doing. Sit with the book open, crouched—or upright, the posture is yours to choose—over a keyboard. You've got an editor brimming with brace-laden code, a console flowing with commands and messages, and inputs and outputs, all surrounded by a flotilla of open browser tabs. Every page of this book is bursting with an invitation to do something—something to try for yourself, play with, puzzle over, be delighted by, and build on.

Fran offers careful and deliberate guidance, filling in the gaps other guides either skip over or get lost in. She presents a simpler path, while ensuring your progress is real. She does not oversimplify or pull any punches—and C++ can summon quite the punch. When you're coding in the zone and your code is working, the going's great;

when things are off, however, C++ can be unforgiving. Fran offers the understanding and support you need.

Good habits—from design to testing—line your route through the language. Constructs and concepts are introduced on an as-needed basis rather than thrown at you in bulk. You will learn and employ language mechanisms, library features, and techniques many experienced C++ developers might consider advanced or be unaware of. The practical flow of this book, however, means they feel appropriate and in service of the problem being solved. No fanfare, no scaremongering, no infodumps, just the right tool for the job.

OK, are you ready? Let's take those first steps!

— *Kevlin Henney*
author of Pattern-Oriented Software Architecture
and editor of 97 Things Every Programmer
Should Know

Preface

You already know how to code, perhaps just a little, but haven't tried C++ yet. You've come to the right place.

C++ is an old language, which has evolved over time. C++ has a reputation of being a difficult language to learn, but I will explain the basics to you. You'll get a deeper understanding of what is happening under the hood in other languages when you take time to learn C++. If you learn C++, you will therefore find many other languages easier to learn. It's worth putting in the time and effort.

There hasn't been a new introductory book recently, so the time had come to write one. There are many older, excellent books, but they don't cover newer language standards. One of my favorites is *Accelerated C++: Practical Programming by Example* by Andrew Koenig and Barbara E. Moo. This is still an excellent resource but doesn't cover newer features, since it was published in 2000.

I'll show you newer C++ features, building up a small program over the course of the book. I mostly target C++23 but give you a glimpse of a few C++26 features too. I also cover older features that haven't changed. I don't have space to cover everything, but you don't need to know everything to get the basics of the language. I'll cover enough to give you a solid foundation.

Who Is This Book For?

I have written this book for people who don't know C++ but have maybe programmed a little in another language. Whether you are a student or a seasoned pro, you will gain something from learning C++. If you have done a little C++ a while ago, this book will help you get back up to speed.

I don't assume any existing knowledge, and I introduce the basics, including variables and loops, near the beginning. You will quickly learn how to use standard C++ features, including containers, to store values, and algorithms to find and sort those values.

You will also learn about classes, which you may have encountered in another language. If you haven't, that's fine. I will explain what they are for and how to use them in C++.

Learning a more functional style, using algorithms, and an object-oriented style, using classes, will give you two different paradigms and more. C++ is very flexible, supporting various approaches, while allowing you to write efficient code. The knowledge you obtain is applicable to other programming languages, so is timeless. You might even want to pursue C++ afterward. My career was in a mixture of languages but predominately C++ in finance, after starting with embedded machines like barcode scanners. I know many others who use C++ in games programming. There are careers out there waiting for you, if you're interested. Or, you can simply learn C++ for the fun of it.

Structure of This Book

There are 15 chapters, starting with a "Hello, world!" program. This gets you started. You need to take a moment to set up your tools to write and build your code. By the end of the first chapter, you will know two different ways to write output to the screen. C++ frequently offers more than one approach, partly because as the language evolves, so do the features. It's worth knowing more than one way to achieve your goals.

From [Chapter 2](#), you will start to build a program you will add to over the course of the book. You will learn how to get input in the second chapter, storing that input in a variable. This introduces the general idea of streams in C++, which you will revisit in [Chapter 8](#) to load data from files. Many industrial-scale programs do not take input directly from a prompt. You would get input from another process, a database or via a graphical user interface (GUI). However, as a beginner it is useful to be able to provide input to your program without having to set up a database or learn another new skill like GUI programming. I also show you how to do some basic testing.

I show you how to deal with problems in [Chapter 3](#). This chapter demonstrates exceptions and the newer `std::expected`. Several people were surprised by these making an appearance so early but decided it was a good idea after reviewing the book. Having thought about testing in [Chapter 2](#), learning how to handle problems in this chapter will help you write good code, taking you beyond knowing the syntax.

Chapters [4](#), [5](#), and [6](#) show you how to use the standard library, starting with arrays and vectors to store elements and moving on to loops, algorithms, and ranges. I also show you lambdas: a way to write a short function directly where you use it. You will start building a trading game in a program that you will add to over the course of the book.

Chapter 7 shows you how to work with random numbers in C++. Knowing how to generate a random number means you can write a variety of games. This chapter will generate random prices, which you will use in the program you have been building.

Chapter 8 shows you how to load prices for your trading game from a file. By this point, you have actually done the hard work, since files are another type of stream, so the code is similar to the second chapter, where you got input from the keyboard.

Chapter 9 does a deep dive into strings; think words or messages. Strings come in various forms in C++, and I show you how to use C-style strings so you can send parameters to your `main` function. People used to claim you needed to learn the C programming language before you could learn C++. This is not true, though C++ did evolve from C and does use some C concepts and types. I will also show you more about using C++ strings, and you will see how much easier they are to work with.

Chapters **10**, **11**, **12**, and **13** show you how to write and use classes. C++ is sometimes described as an object-oriented (OO) language, meaning you write code based on a *class*: a way to group together variables and functions. You can write OO code in C++, but you don't have to. These chapters require more details about how the language works and how to think about your design.

Chapter 14 shows how to use a *variant*: a type that can be used for one of a fixed set of types. This newer feature can be used in various ways. I'll also show you how to use the `std::visit` function to work with the various types in a `std::variant`. Together these form a newer approach to writing C++, and I believe they are a vital new feature.

The book finishes with a look at templates and lookup tables, in the form of a `std::unordered_map`. Templates are a big topic and are very powerful. I only have space to give a short introduction, but it is enough to help you understand what you need to do to write your own template. By this point you will have covered a lot of C++, but not everything. You will have enough knowledge to continue your journey if you so wish. There will be more to learn, and C++ will continue to evolve. Knowing the basics will give you a solid grounding.

Try to write the code as you read. Play with the trading game you produce, and note any questions you have as you read. Find someone to talk to if you get stuck. Find someone to share with if you understand something new. Above all, learn lots and have fun.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/doctorlove/IntroducingCpp.git>.

If you have a technical question or a problem using the code examples, please send email to support@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Introducing C++* by Frances Buontempo (O'Reilly). Copyright 2026 Frances Buontempo, 978-1-098-17814-7.”

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning

O'REILLY® For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
<https://oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/introducing-c-1e>.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly>.

Watch us on YouTube: <https://youtube.com/oreillymedia>.

Acknowledgments

I'd like to thank everyone who helped me write this book. First, thanks to my editor Sarah Grey, who faithfully supported me while writing and kept fixing my typos, and to Klaus Iglberger, who originally suggested I could write a new introductory C++ book and diligently helped by providing feedback on each chapter. You have both helped me write a better book than I could have managed on my own.

Next, I'd like to thank Kevlin Henney, both for giving me early feedback and for taking time to write a foreword. Finding a few moments to chat with you early on, when I started writing, was very helpful. Thanks for your encouragement.

Finally, thank you to all my reviewers, including Herb Suter, Daniela Engert, Emyr Williams, Scott Furry, Danny Faught, Chris Jenkins, Jess Males, Robin Rowe, Ben Reed, Berill Effi Farkas, Matt Godbolt, Andreas Fertig, and Steve Love. Your feedback let me know how others might read my words, helping me to be clearer. Any mistakes left are entirely my fault.

Hope you enjoy reading this book and go on to do great things.

Hello, World!

Humans write code, but computers understand only 0s and 1s, so code needs to be “translated” for the computer. Some languages, like Python and JavaScript, are *interpreted*, meaning that the tools read the code and decide what to do *dynamically*, meaning at runtime. Every time such code is run, it must be reinterpreted.

Other languages, like Java and C#, *compile* to an *intermediate language*, for example, bytecode for Java. The output is interpreted by a virtual machine, so you can compile your code once and run it almost anywhere. This is more efficient than interpreting code every time it is run, because the initial transformation step needs happen only once.

C++ is different, though it follows a process used by C, Fortran, and several other languages. C++ source code is transformed *directly* into something the computer understands. When you build C++ code, two steps happen.

First, a *compiler* reads your code and produces object files that are specific to your target machine, such as 32-bit Windows, 64-bit Linux, or an embedded system. If you want your code to run on a different machine, you need to rebuild it. For a small program, the object files produced by the compiler might stay in memory; for a larger program, you are likely to see them, often with the extensions **.o* or **.obj*, generated on your machine.

Second, the *linker* stitches the object files together to produce a library, which you can use in another codebase, or program that you can run directly.

In short, you write code, and the compiler parses your code and generates object files, which the linker joins together into a program you can run (or a library you can use), as shown in [Figure 1-1](#).

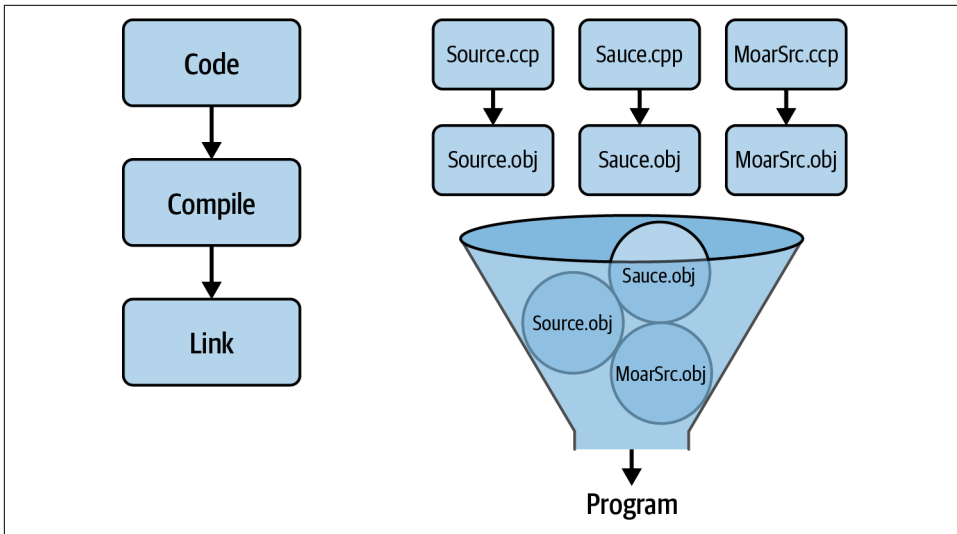


Figure 1-1. The compiler uses source files to generate object files, and the linker pieces these together to make the final output

In theory, compiling and linking up front for a specific machine can make a program run quicker. In fact, C++ is often chosen for speed; your browser or Java virtual machine may be implemented in it.

C++ does have a reputation of being difficult. It is a relatively low-level language, which gives you more control and the potential to write very fast code. To draw an analogy, a car with a manual transmission gives the driver more control than an automatic. You can go faster, but if you don't know what you're doing, you might use the wrong gear or stall the engine.

Similarly, one small mistake in your C++ code could make the compiler spew forth many errors or make the linker simply claim, “Error, function not found.” You might find you hit a problem once in a while as you work through this book. Don't panic. You will build up an intuition of where to look for problems, and I will guide you, starting simply.

If you already know another programming language, you will get more of a feel for what happens under the hood in that language as you read this book. C++ takes you closer to the hardware, giving you a deeper understanding of programming in any language. Taking time to learn C++ will pay off.

In this chapter, I will walk you through a very short code example to print a greeting on the screen. I will explain some background and syntax, giving you code to try. You will learn about the `main` program entry point and the basics of a function, and you'll get your toolchain up and working.

To get the most from this book, try to run the code examples and play with them. By the end of this chapter, you will be able to build and run a small program that generates output, and you'll know some basic C++ syntax. You will then be prepared to handle input in the next chapter.

Installing Tools

You can use Vim, Emacs, Notepad++, or another editor to type in your code. You then need to build your code. Alternatively, you can use an integrated development environment (IDE) to do both.

After you choose an editor, you need C++ tools to build your code. Your machine might already have some tools installed. People often refer to a “C++ compiler” when they really mean a compiler and a linker. If I slip into saying “compiler,” you’ll know I mean both.

Another useful tool is the **Compiler Explorer**, which lets you try out code online. People often refer to the site as Godbolt, because it was created by Matt Godbolt. You can choose a compiler and build and run your code there. By default, you type your code in the left window, and the right-hand side shows the assembly output. The linker uses the assembled output to form the program, as shown in **Figure 1-2**.

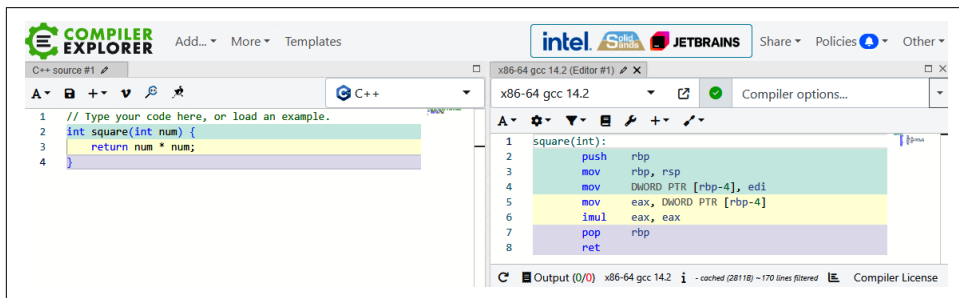


Figure 1-2. The default landing page on Compiler Explorer, showing code on the left and assembled output on the right

Let’s see if you have C++ tools already and install some if you don’t. I will tend to use C++23 in this book, so you might have older tools installed and need to upgrade.

Linux and macOS

Linux tends to come with the **GNU Compiler Collection (GCC)**. GCC can build several languages and provides a tool called `g++` to build C++ code. See if you have `g++` by opening a prompt and typing:

```
g++ --version
```

If you see a version number, you have a compiler. However, if the version number is not in double figures, it's very old. Different versions of the tools support different versions of C++. This book will show you newer features. Find your system's instructions to install GCC or get a newer version, if needed.

macOS tends to have **Clang** installed. Clang comes with various tools, including clang++ for C++. Clang uses the **LLVM** compiler and toolchain. Open a prompt and try:

```
clang++ --version
```

If you get a version number that is not in double figures, it's also very old. Again, find your system's instructions to install Clang or get a newer version, if needed.

Windows

For Windows, you also have a choice: you can try Clang, GCC, or the Microsoft compiler. Microsoft provides **step-by-step instructions** for all three of these toolchains. Alternatively, you can get the community edition of **Visual Studio**, which provides the Visual Studio C++ tools and IDE.

A prompt needs to know the path to the tools, so you can either manually add this to your PATH environment variable or open a Developer Command Prompt if you've installed Visual Studio. From a suitable prompt, type `cl`, and see what it says:

```
> cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.40.33811 for x86
```

If you see an older version, you need to upgrade.

Using Your Tools

Let's make a program you can run. **Figure 1-1** shows several source files being built together, but you need only a single file to make your first program.

C++ programs are composed of *functions*, which group statements into a block. Functions can call other functions. A function may or may not return a value. Functions use the keyword `void` to indicate they don't return anything.

When you write a function that returns a value, you must specify its type. All values in C++ have a type. For example, the basic numeric type is `int` (integer), which can represent both positive and negative numbers. The maximum and minimum values of an `int` depend on the target machine and compiler, but the C++ standard guarantees a range of at least $-32,768$ to $32,767$. Most modern machines have a much larger range, often $-2,147,483,648$ to $2,147,483,647$. Because `int` is a **fundamental type**, it is immediately available as part of the *core language*.



C++ is standardized by the International Organization for Standardization (ISO). A working group, called WG21, agrees on new versions. The **ISO Cpp** has details about the process. There are other working groups for other languages; however, many languages are not ISO-standardized.

A program or app requires a function named `main`, which always returns an `int`, indicating success or failure. Create an empty file, call it *empty.cpp*, and type in the very short program shown in **Example 1-1**.

Example 1-1. Main C++ function

```
int main() ❶  
{ ❷  
} ❷
```

❶ Function head

❷ Function body

Almost any C++ function has a return type, a function name, and parentheses indicating any *parameters*, or values, sent into the function. Empty parentheses mean the function has no parameters.

There are a few different ways to introduce a function. However, let's start simply. You need a return type, a name, and empty parentheses. This forms a *function head* or *signature*, which looks like this:

```
int main()
```

After the function head is a set of curly braces. The statements comprising the *function body* go in between these, but an empty function is fine too (though it won't do much).

The function `main` is special. By default, it returns `int 0` to indicate no errors, so you don't need to specify the return in the function body. Also, you can have only one `main` function in your program.

Save your file, and you're ready to build your first C++ program. Open a command prompt and navigate to your *empty.cpp* file.



I am going to explain two important flags for you to provide to the toolchain and show you how to set them from a prompt. If you want to use an IDE instead, look at your IDE's documentation. It's OK to use an IDE, but if you try the command line, it will help you to remember what is happening.

The instructions vary slightly between compilers, but all have parts in common, following this pattern:

```
tool_name [optional flags] source_name.cpp -o output_name
```

You state the tool (for example, clang++), maybe use some flags, and then state the source file or files. You can use `-o` to specify the output program's name. If you don't specify this, by default, g++ and clang++ produce a program called `a.out`. Windows uses a different convention for specifying the output name but will pick a sensible default name based on the input file name.

You are going to use two optional flags. First, you will ask for (almost) all warnings. Clang and GCC use `-Wall` for warnings, and you should use `/W4` for Windows. You can ask for extra warnings too. (See [C++ best practices](#) for details on Clang and GCC.)

Most languages have warnings. They indicate a potential problem, rather than a mistake. A warning is not an error, but it might be telling you something important. In C++, you can turn on extra warnings, too. For now, using `Wall` is enough.

The second optional flag you will use states which version or standard of C++ you require. For example, for C++23, you might use `-std=c++23`. For Windows, use a slash instead of a dash and a colon instead of an equal sign `/std:c++23`. For older versions of Clang, use `2b` instead of `23`: `-std=c++2b`. Without the flag, each compiler defaults to a different version. If your compiler claims `-std=c++23` or `-std=c++2b` is not supported, refer back to [“Installing Tools” on page 3](#) and upgrade to a newer compiler.

Each compiler can target a specific range of language versions. A new version of C++ has been released every three years since 2011, and the number indicates the year. Each compiler version tends to implement a subset of the full standard, so if a feature may not work for you, I'll let you know and suggest an alternative approach.



The [C++Reference website](#) lists which compilers support which features. Have a look if you see errors telling you your compiler doesn't believe a certain function exists, or similar. Remember to use the `std=` flag.

The instructions for building your source file into a program vary slightly between tools, so follow the appropriate subsection to build your code.

GCC

If you are using GCC, open a prompt and type:

```
g++ -Wall -std=c++23 empty.cpp -o empty
```

If you have more than one compiler version, you can specify a specific tool: for example, `g++-14` instead of `g++`.

Clang

For Clang, open a prompt and type:

```
clang++ -Wall -std=c++2b empty.cpp -o empty
```

If you have more than one compiler version, you can specify a specific tool: for example, `clang++-15` instead of `clang++`.

Windows

For Windows, you need the tools on your path. If you have a version of Visual Studio installed, the simplest way to do this is to open a developer command prompt from the Windows start menu.

Windows uses slashes instead of dashes, and you use another optional flag, `/EHsc`. This enables standard exception handling. You'll learn about exceptions later, but without this flag, you will get lots of warnings:

```
cl /W4 /std:c++latest /EHsc empty.cpp
```

You haven't specified the output, so Windows chooses *empty.exe* for you based on the source filename.

Running Your Program

Run your program using `.\empty.exe` on Windows or `./empty` on a Mac or Linux. You won't see much, because the code doesn't do anything. However, you are now set up and ready to learn more. You can run the program as often as you want now, without needing to recompile and link the original source code.

Congratulations—you've compiled and linked your first program! Now let's make it actually do something.

Writing to the Screen

You are now going to print "Hello, world!" to the screen. There is more than one way to do this in C++. I will show you two approaches here.

I noted that C++ has a new version every three years. C++23 introduced the function `println`. However, your toolchain might not support it. If this approach does not work for you, seeing the error output is useful, and you'll learn an alternative approach shortly.

As you work your way through this book, you might find that some functions are not yet available in your compiler. If this happens, try using the Compiler Explorer.

Using `println`

The `println` function is part of the *standard library*. The standard library comes with your toolchain. You can use standard library facilities by including an appropriate header—in this case, `print`. You used `int` earlier, which I noted is part of the core language so doesn't need an `include`.

Recent versions (as I write this in late 2025) have introduced a new approach using modules, but many existing codebases still use the older method including headers. Modules can be harder to get working, so using headers is easier.

Create a new file called *hello_println.cpp*, either in your IDE or in an editor, and add the `include` line and the empty `main` function as follows:

```
#include <print> ❶

int main() ❷
{
}
```

- ❶ Includes the `<print>` header
- ❷ Defines an empty (for now) `main` function

If you include the items you want to use near the top of your file, you'll be able to find them easily. You'll usually see standard headers specified in angle brackets. Later, you'll include your own headers using double quotes.

This code is similar to the first, with no code in [Example 1-1](#), apart from the `include` line, so it won't do anything yet. Add your greeting using the `println` function from the `print` header. There are various ways to use `println`, but the simplest takes a single message in double quotes, like "Hello, world!"

Type the code into your *hello_println.cpp* file and save it, as shown in [Example 1-2](#).

Example 1-2. Using `println`

```
#include <print>

int main()
{
    std::println("Hello, world!"); ❶
}
```

❶ Prints a message

You have added a single line to the function body. Let's think through what this line does. As with [Example 1-1](#), `main` returns an `int` of 0 for you, so you don't need to explicitly return a value. The `println` function prints the message on a line and adds a newline character (`\n`) at the end. If you send messages to the screen, you might want further messages to start on the next line, so you'll need a newline character at the end. The `println` function adds this for you.

Notice that the `println("Hello, world!")` instruction ends with a semicolon. In C++, every statement must be followed by a semicolon. This means that spacing isn't significant. Some languages, like Python, use indentation and whitespace to denote blocks, but not C++. The combination of semicolons to end a statement and braces to indicate where groups of code start and stop mean that whitespace is insignificant. However, people usually indent the function body (the contents between the curly braces) to make it stand out visually.

You need to put a function's code inside its braces, which form its *block scope*. The concept of *scope* will crop up in various ways over the course of this book. In simple terms, the scope groups code together and allows magic to happen at the end of the block.

The `print` header also uses a different kind of scope: it groups code into a space with a name, called a *namespace*. All standard library facilities live inside the standard namespace, spelled `std`. You can prefix the function you need with `std::` to indicate where it comes from. This prefix consists of the name `std`, followed by the *scope-resolution operator*, `::`. You're free to create your own namespaces, and doing so means you can specify where the compiler can find the function you want to use. Without `std::`, the compiler and linker will look for a function called `println` *outside* the standard namespace—and won't find it. This leads to errors from the linker along the lines of `unresolved symbol`.

Save your file and build your code, using the warning and `std` version flags. Either find the “build and run” option in your IDE or use a prompt, as you did before, with the appropriate command for your toolchain:

```
g++ -Wall -std=c++23 hello_println.cpp -o hello_println
```

```
clang++ -Wall -std=c++2b hello_println.cpp -o hello_println
```

```
cl /W4 /std:c++latest /EHsc hello_println.cpp
```



If you are using an IDE or visual editor, you can find instructions in its documentation for setting warning flags and the `std` language version.

If that worked, run your program. You should see a greeting on the screen:

```
Hello, world!
```

Take a moment to look back at the code. You added an `include` to a standard library header. You used only one function here, but the `<print>` header contains more.

You can check [CppReference](#) to see what's available. You will see various versions of `println`, including formats for aligning code and files, which I'll show you later. There is a `print` function, too, which doesn't add the newline character. Try this in your code instead of `println` and see what happens.

Troubleshooting

Your code might fail to compile. You might see a message like this:

```
hello_println.cpp:1:10: fatal error: print: No such file or directory
  1 | #include <print>
    |           ^~~~~~
compilation terminated.
```

Or this:

```
hello_println.cpp:1:10: fatal error: 'print' file not found
#include <print>
      ^~~~~~
1 error generated.
```

Or this:

```
\include\print(11): warning STL4038:
  The contents of <print> are available only with C++23 or later.
hello_println.cpp(5): error C2039: 'println': is not a member of 'std'
predefined C++ types (compiler internal)(346): note: see declaration of 'std'
hello_println.cpp(5): error C3861: 'println': identifier not found
```

If you see such a message, your toolchain does not support the `print` library features yet. The next section shows an older approach that will work.



In general, if you see a message saying that some feature in the namespace `std::` doesn't exist, your toolchain may not support the feature you are trying to use. You can use the Compiler Explorer instead. For example, try a [g++-14 build](#) of the “Hello, world!” code that does compile.

Using cout

Before C++23, C++ provided something called `std::cout` to print output instead of `println`. Many codebases still use this approach, and input uses a similar method, so the older way is worth learning and provides a useful starting point for input in [Chapter 2](#).

This standard library feature is declared in the `iostream` header. The object `std::cout`, pronounced “see out,” is a global object associated with a *standard C output stream*. A *stream* in C++ is a sequence of characters. You may have used a stream processor before, such as `awk` or `sed`. `cout` handles character output. [American Standard Code for Information Interchange \(std::cout utf-8 ASCII\) characters](#) work on all platforms, and `std::cout` can handle Unicode characters, but you may need to change some settings to make non-ASCII characters work on [Windows](#).

Create a new file called `hello.cpp`. Add the `include` line and the `main` function, as you did before. This time, include the `<iostream>` header and use the `std::cout` stream insertion operator `<<` to print the greeting, as shown in [Example 1-3](#).

Example 1-3. A greeting, using `std::cout`

```
#include <iostream> ❶

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

❶ Includes the `iostream` header

❷ Prints a message

As before, you have to put the statement inside the `main` function’s braces and end the line with a semicolon. Putting some whitespace at the start of the line is conventional, but not required.

Build this single file. If you can’t remember what to do, refer to the command-line instruction after [Example 1-2](#), changing the filename and output. Don’t forget to save your file and use the warning and `std` version flags.

When you run the new version, you will see that there is no new line after the greeting. When I run the code from a prompt, I get the following:

```
./hello
Hello, world!$
```

The greeting is displayed, and my prompt symbol appears on the same line, immediately after the message. In contrast, `println` prints the greeting *and* a new line. If you use `std::cout`, you need to ask for a new line if you want one. The simplest way to do this is to use `\n`. The backslash *escapes* the `n` for a new line, meaning it indicates that the character after the backslash has a special meaning. There are several other escape sequences, including `\t` for tab and `\\` for a single backslash.

You can add the extra character to the message, as shown in [Example 1-4](#).

Example 1-4. Using `cout` with a new line

```
#include <iostream>

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

When you build and run this code, your prompt starts on a new line:

```
$/hello
Hello, world!
$
```

Understanding `println` and `cout` in Depth

You should have at least one “Hello, world!” program working now. Before moving on, let’s take a deeper look at what you’ve done so far.

You started with a `main` function:

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

As with most functions, it has a return type and parentheses, along with a name. A function head followed by a semicolon *declares* the function; that is, it tells the compiler that this function exists somewhere. The function *definition* is code in curly braces. The standard headers often contain the declarations only but sometimes have the full definition.

You used two functions from the standard library. First, you used `println`. There are several versions (or *overloads*), but they all have a similar signature:

```
void println();
void println(/* maybe some parameters*/);
```

An *overloaded function* has the same name but takes different parameters. For example, you can use `println` with no parameters to print a blank line or use the second overload to print a message and a new line.

Recall, `main` returns an `int`. `main` is special: it will return a 0 by default, and you can have only one `main` function in your program. The `println` function definition starts with `void`, meaning that it doesn't return anything.

You also used `cout`, along with the stream insertion operator `<<` to display "Hello, world!". There is an overload for `int` and other types. CppReference gives a [long list of overloads](#). This website is a great place to look up details. Trying to search for punctuation like `<<` can be hard, but knowing it is an operator helps. An *operator* is a special type of function made up of a symbol, like `+` or `<<`. You can use the symbol between the *operands* or *arguments*; that is, the specific parameters that you want to apply the operator to. They can make code more readable. For example, it's more natural to say `1+2` than `operator+(1, 2)`.

You added the `\n` character inside the greeting "Hello, world!\n", but you have an alternative. The stream insertion operator returns the same stream you started with, so you can *chain* calls together. You can also chain operations; for example, you can chain additions to calculate `1 + 2 + 3`. To append the newline character, use `<<` again, with single quotes for the single character:

```
std::cout << "Hello, world!" << '\n';
```

The compiler parses the statement from left to right. The leftmost argument is the message, so that is used first. Then the `<<` operator is applied a second time with the newline character:

```
(std::cout << "Hello, world!") << '\n';
```

You don't need parentheses here, since the two statements are equivalent.

You can chain lots of different messages together. You might find that `std::cout` writes out very long messages in chunks. Since it takes a while to render the characters on a screen, it buffers them and writes a few at a time.

You will sometimes see `std::endl` used instead of `\n`. That does two things:

- Appends a new line
- Flushes the buffer

Flushing the buffer ensures that everything in the buffer is written. If a program crashes without doing this, any buffered characters may never make it to the stream. `std::endl` is a helper function to control a stream, called a *manipulator*, and using it is equivalent to the following:

```
std::cout << '\n' << std::flush;
```

You are unlikely to need to do this, but if your code crashes, you might not see all the output unless you have flushed the buffer beforehand. You may come across other learning resources telling you to use `std::endl`; now you know that you have a choice.

Conclusion

This chapter covered the difference between interpreted and compiled languages. C++ is compiled. The compiler reads the source code, parsing it into object files, and then your toolchain links the object files together into a program.

You also set up your C++ toolchain and built some code. You wrote one function (a few times):

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

You also used two functions from the standard library, `std::println` and operator `<<` with `std::cout`. To use these, you included the appropriate headers. Most of the output in the rest of this book will use `std::cout`, but you will revisit `std::println` in [Chapter 9](#). C++ often has more than one way to achieve something. Knowing the alternatives will help you understand the language better.

You covered some important ideas:

- Functions have a head and a body; the body goes inside curly braces.
- Statements end in a semicolon.
- `main` is special and where a program starts.
- `int` is a built-in type for positive and negative numbers.
- `void` indicates that a function returns nothing.
- `std` indicates the standard namespace.
- `::` is the scope-resolution operator.

You have written your first C++ program. You've learned some basic syntax and how to call some standard library functions. You've written output, so you're probably wondering: how do you take input? Let's find out.

Variables and Keyboard Input

In this chapter, you will accept input in another short program. You will learn about declaring variables and practice writing more functions. You'll also start to think about general approaches to handling errors, which I'll go into more in the next chapter. You'll then be ready to start building a larger program in [Chapter 3](#). The larger program will input and analyze stock-price data, and you will add to this project over the rest of the book.

First, you need to be able to take input, so let's find out how. You will write a program in a single file again. If you need a reminder on how to build your code, look back at [“Using Your Tools” on page 4](#).

When you wrote output, you used two approaches: `std::println` and `std::cout`'s operator `<<`. You have one option for input in C++ using an input stream `std::cin`. Input needs to go somewhere, so you must start with a place for it.

Declaring Variables

Create a new source file and call it *input.cpp*. This will give you a place to experiment. You will write a program to get input shortly, but first, you need a variable to take that input. All variables in C++ have a *type*. You met `int` in the previous chapter as the return from `main` in [Example 1-1](#). You start with a type followed by a name. You can explicitly state a value you want. For example:

```
int number=0;
```

However, a more general approach uses *brace initialization*, which means using `{}` after the variable name:

```
int number{};
```


The `number` is still initialized with a zero. This modern approach was introduced in C++11, so don't forget to use the `std` flag when you build code using this. You can put a number in the braces, like `{0}`, but you don't need to if you want a zero. The ISO [Core Guidelines](#) gives more details on this `{}`-initializer syntax.



The ISO [Core Guidelines](#) are an open source collection of guidelines edited by Bjarne Stroustrup, the inventor of C++, and Herb Sutter, a prominent C++ expert. The guidelines aim to help C++ programmers to write simpler, more efficient, more maintainable code.

Since `number` is a *variable*, you can vary its value, for example, by changing it later:

```
number = 42;
```

C++ allows you to flag a variable as `const`, short for *constant*, meaning you do not intend to change it from the initial value. If you try to, you'll get an error. The following does not compile:

```
const int number{1};  
number = 73;
```

You will see more uses of `const` over the course of this book, but you want to change `number` based on input, so you won't use `const` here.

In your `input.cpp` file, declare an `int` inside the `main` function:

```
int main()  
{  
    int number{}; ❶  
}
```

❶ Declares a number and uses `{}` to initialize it to zero

After saving your file, check that this builds OK, using the `warning` and `std` flags. If so, you're ready to get some input. The `warning` flag will cause a compiler *warning* pointing out that you have an *unused variable*. That's a good thing—remember, a warning is not an error. However, warnings often point out something you need to think about, so watch for them. You will use the variable shortly, so the warning will stop.

Character Input

You obtain input from `std::cin` (pronounced “see in”), which lives in the `<iostream>` header along with `std::cout`. Output uses the operator `<<`. Input goes the other way, so it uses the operator `>>`, called the *stream extraction operator*. Any leading whitespace, such as a space or tab, will be ignored. You'll signal the end of

your input by pressing Enter (or Return). You can type any characters until Enter, but whitespace will break the input into separate parts, as indicated in [Figure 2-1](#).

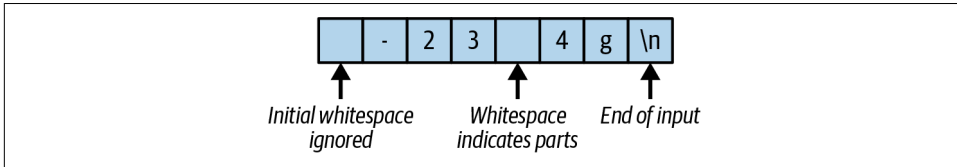


Figure 2-1. Input is a stream of characters, split into parts by whitespace

When you extract input, you store it in a variable, which is why you started by declaring a number in the previous section. You can try to extract other types too, but let's start with an `int`.

Add code to your `main` function in `input.cpp`, as shown in [Example 2-1](#). You need a variable to store the number, code to get the input, and code showing what the input was. Adding a helpful message asking for numeric input first tells anyone using the program what to do. Using a `>` symbol makes it clear the program is prompting for input.

Example 2-1. Attempt to input whole numbers

```
#include <iostream> ❶

int main() ❷
{
    std::cout << "Please enter a number.\n>"; ❸
    int number{}; ❹
    std::cin >> number; ❺
    std::cout << number << '\n'; ❻
}
```

- ❶ Includes the input/output stream header
- ❷ Starts the `main` function
- ❸ Shows a helpful message, ending with a newline and the `>` character to prompt for input
- ❹ Declares a variable and initializes it
- ❺ Tries to stream in a number
- ❻ Streams out the number

You saw `std::println` and `std::cout` in [Chapter 1](#). I will stick with `std::cout` in this chapter, because it shows the parallels between the two stream operators. You use `<<` to stream things out, and `>>` to stream things in.

Build your code and try it out. At this point, you probably don't need me to remind you to save your file and use the appropriate flags.

You will see the message and the prompt to enter a number:

```
Please enter a number .  
>
```

Nothing further will happen until you type something and press Enter.

Try some positive and negative numbers. Get inventive: try fractions or pretend your cat walked over your keyboard. (If you have a cat, you know this can happen.) You can change the type from an `int` to a more general numeric type to deal with numbers that have a decimal part. You'll do that shortly.

When you deal with input, things can go wrong, so the next section introduces some considerations.

Detecting Input Problems

Your program is trying to obtain a number from the keyboard. The stream of characters the user types might not be a number, though. What happens if you don't type in a whole number, or even a number at all?

Input of Real Numbers

Let's consider numbers with decimal parts first. When you used `std::cin >> number;`, you asked for the `int` *overload*, or version, of operator `>>`. If you tried a real number, like `-1.4`, rather than a whole number, the output picked up only the whole-number part: `-1`. The `.4` will be left in the stream. You can find it later or ask the stream to ignore these characters, which I will show you how to do shortly.

A simpler approach is using another numeric type, a `double`, which is suitable for *floating-point* values. (This type's full name is *double-precision floating point*.) As a reminder, integers are whole numbers, like `-1`, while doubles can have digits after the decimal point, so they include integers as well as numbers, like `-1.4`.

Try changing the type of number in your main function, as shown in [Example 2-2](#), and see what happens.

Example 2-2. Input of doubles

```
#include <iostream>

int main()
{
    std::cout << "Please enter a number.\n";
    double number{}; ❶
    std::cin >> number;
    std::cout << number << '\n';
}
```

- ❶ Declares a double instead of an int

As before, you'll be prompted to enter a number:

```
Please enter a number.
>
```

Now you can enter numbers with a decimal part. Both the `double` and the `int` are **fundamental types**, and the exact range of values they support can vary between tool-chains. You can find out the largest available value by using a function called `max` in the `<limits>` header. The `max` function comes from a *class template* called `std::numeric_limits`. You've already learned that classes, like `std::cin`, group functions together. Templates are like cookie cutters or patterns for code of various types. They're like Java or C#'s generics, but much more powerful. So, a class template is a pattern for making classes.

You provide the template's type in angle brackets and then call the `max` function using the scope resolution operator `::`, like this:

```
int largest_int = std::numeric_limits<int>::max(); ❶
double largest_double = std::numeric_limits<double>::max(); ❷
```

- ❶ Asks for `int`'s maximum
- ❷ Asks for `double`'s maximum

You have used the scope resolution operator to use types and objects from the standard library like this: `std::`. You also call functions declared in a type, called *static member functions*, using `::`. These static functions are available from the type itself.

Detecting Problems with Numbers

Now, since both `int` and `double` have a minimum and maximum value, if you (or your cat) press a digit for several seconds, you can end up with a number that's too big for its type. The extraction operator will use as many characters as fit in the type, leaving unused characters in the stream's buffer. I will show you how to check if

characters are left over, and you will learn some more C++ syntax on the way. Then, in the next section, I'll show you a simpler way to deal with invalid input. Try the code here in a file called *input_experiment.cpp*. Add a `main` function for the code.

Input streams, including `std::cin`, have an `eof` function, which stands for *end of file*, which tells you if there are more characters left in the stream after an attempt to read. (Files are another type of stream you will meet later.) You can check for `std::cin`'s `eof` after the read, using the dot operator:

```
if(!std::cin.eof())
{
    std::cout << "Unused input\n";
}
```

`eof` is a member function of the input stream, so it is part of `std::cin` itself. Member functions need to be called via the dot (period) operator. You used `::` to call `numeric_limits`'s static `max` function earlier. Now you are calling an *instance* function, so use a dot. The `!` symbol means not, so the `if` checks whether you have reached the end of the stream. When there are further characters waiting in the stream after the number has been read, `std::cin` is *not* at `eof`. If you type in a smaller number and press Enter, the `\n` character will still be in the stream, so valid input will also not be at the `eof`. You can take a peek at the next character, using `std::cin`'s `peek` function, and compare it with a new line:

```
if(std::cin.peek() != '\n') ❶
{
}
```

❶ Compares the next character with a new line using `!=` for inequality

So you need to check two things. You can use the symbol `&&` to check that something *and* something else are both true:

```
if(!std::cin.eof() && std::cin.peek() != '\n')
{
}
```

Add this to check the `main` function in your *input_experiment.cpp* file, and try some values to test it. Add the maximum possible value to the prompt for input so you know what your toolchain supports, as shown in [Example 2-3](#).

Example 2-3. An attempt to spot input problems

```
#include <iostream>
#include <limits>

int main()
{
```

```

const double largest = std::numeric_limits<double>::max(); ❶
std::cout << "Please enter a number up to " << largest << ".\n>"; ❷
double number{};
std::cin >> number;
std::cout << number << '\n';
if(!std::cin.eof() && std::cin.peek()!='\n') ❸
{
    std::cout << "Unused input\n"; ❹
}
}

```

- ❶ Finds the maximum double
- ❷ Prompts for input
- ❸ Checks if there is input left over which isn't a newline character
- ❹ Reports that something bad happened

When you build and run this, you will see the maximum in **scientific notation**, like this:

```

Please enter a number up to 1.79769e+308.
>

```

Typing a single number smaller than the maximum is fine. If you type a few numbers with spaces, the program picks the first and then reports that there is leftover input:

```

Please enter a number up to 1.79769e+308.
>4 5 6
4
Unused input

```

Looking good so far. If you type a very large number, like 1e+309, you might see the message `Unused input`, and the output may show a number, like this:

```

Please enter a number up to 1.79769e+308.
>1e+309
1.79769e+308
Unused input

```

That number is wrong! To be fair, the value shown is the largest possible double. Some versions of clang++ don't show `Unused input`; instead, they show the value `inf` (meaning infinity). Whatever you see, the number is still wrong. So let's find some better ways to handle input problems.

Detecting More General Problems

If you type a digit or two and then some letters or other characters, the program will take as much from the stream as possible for the number. If I type a digit and a letter, I get this:

```
Please enter a number up to 1.79769e+308.  
>1a  
1  
Unused input
```

What happens if you just type a letter and then press Enter? Think it through before you try.

Now, `double number{};` sets the variable to zero. The extraction `std::cin >> number;` will stop at the Enter, so `number` will remain at zero, and the letter will be left over in the input. If you run this, you therefore get:

```
Please enter a number up to 1.79769e+308.  
>q  
0  
Unused input
```

This situation isn't the same as having too *many* digits, but you can solve these problems in the same way. This program expected numeric input but got something else. Along with the `eof` function, a stream has a `fail` function, which tells you if its extraction has gone wrong:

```
if(std::cin.fail())  
{  
    std::cout << "Something went wrong!\n";  
}
```

In general, lots of other things could go wrong. For completeness, the stream also has a `bad` function that indicates if any other bad things have happened. CppReference gives a long list of [possible problems](#). Fortunately, you don't need to know about all of them. Rather than checking `fail` and `bad` directly, C++ gives us a neater way to check that you're good, like this:

```
if(std::cin)  
{  
    std::cout << number << '\n';  
}  
else  
{  
    std::cout << "Bother!\n";  
}
```

The `if(stream)` is shorthand for `if(stream.operator bool())`, which is a bit of a mouthful. Such shorthands are called *syntactic sugar*. An `if` statement checks a conditional expression, so the expression needs to give a `bool`. `std::cin` has an operator

returning a `bool`, so the stream can be *converted* to a `bool`, allowing you to use it in a *Boolean context*, for example, inside an `if` statement. The operator `bool` checks if the stream is good, so a `fail` or `bad` state gives `false`. In effect, the syntactic sugar saves you the work of explicitly checking the state.

The operator `bool` is provided for use in a Boolean context only. You can't assign the stream to a `bool`:

```
bool ok = std::cin;
```

You are trying to set a `bool` to a stream. A stream isn't a `bool`, so if you try this line of code, you will get a compiler error. **Godbolt** using Clang gives this error:

```
<source>:11:10: error:
no viable conversion from 'istream' (aka 'basic_istream<char>') to 'bool'
  11 |         bool ok = std::cin;
      |         ^~~~~~
/./././././include/c++/15.0.0/bits/basic_ios.h:121:16:
note: explicit conversion function is not a candidate
  121 |         explicit operator bool() const
      |         ^
  1 error generated.
Compiler returned: 1
```

Other compilers will return a similar error.



Some compiler error messages are long and might seem intimidating. Don't panic! Just focus on the words you do understand and the line numbers it calls out.

Operator `bool` is mentioned in the message, but you also see a `const` and an `explicit`. You saw `const` earlier in “**Declaring Variables**” on page 15. Here, it's used on a stream's function, meaning it does not change the stream itself.

The keyword `explicit` means you can use a function or operator only in specific places. So, using operator `bool` inside an `if` statement or another place that needs a `bool` is fine, but you can't assign the stream to a `bool` directly. The `if` calls the operator `bool` for you. The thing to remember is that an `if` on `std::cin` is checking that things are OK.

Pulling together what you have learned so far, you now have a small program that attempts to get numeric input and detects problems, as shown in **Example 2-4**.

Example 2-4. Input numbers and check for problems

```
#include <iostream>
```



```

int main()
{
    std::cout << "Please enter a number.\n>";
    double number{};
    std::cin >> number;
    if(std::cin) ❶
    {
        std::cout << number << '\n';
    }
    else
    {
        std::cout << "Something went wrong\n";
    }
}

```

❶ Treats the stream as a `bool` to check if it is OK

This code will extract a number if possible but leaves unused input behind. Entering a single number isn't very interesting, but now you know more C++. You will see how to extend the code to input several numbers in [Chapter 4](#). For the moment, though, let's improve the code to deal with problems.

A Function for Input with Some Tests

Putting code in `main` makes it harder to test. Let's move the input into its own function and call that function from `main`. You'll also see a basic way to test code. Create a new source file called *input_with_tests.cpp* for this section.

You have written only one function so far, `main`, but you've used several from the standard library. You know that most functions have a return type, a name, and a set of parentheses to hold parameters sent into the function. If you write a function to get numeric input, what should its signature be? You haven't written any tests yet, and writing tests first can be a good way to think about designing code (this is called test-driven development, or TDD). C++ doesn't come with a unit-testing framework, though several external libraries are available. To keep it simple, you can use the library function `assert` to check values.

Starting with a Failing Test

Since you're putting code in one source file, let's add a test function there too. If you're used to unit testing, you know that you'd normally split out the tests from the code. You can do that with a testing framework, but here, let's do the simplest thing that works.

Start with an empty function in a file called *input_with_tests.cpp*, and call it from `main`. The test function will call `assert`, which comes from the `<cassert>` header.

(The `c` at the start tells you that this header was originally part of the C standard library.) Start with this code:

```
#include <cassert>

void test_code()
{
}

int main()
{
    test_code();
}
```

This doesn't do much, but it does mean you can now add some tests.

Note you have written the `test_code` function before `main`. If you call a function before the compiler sees a declaration or definition, you'll get an error. That's because the compiler needs to know about the function before you can use it. Alternatively, you can put the declaration first and then define the function near the end. You will learn more about this approach when you write your own header file in [“Getting Several Numbers Into a Vector \(Again\)”](#) on page 83.

Let's write a test. An `assert` checks a conditional expression, such as `0==1`, and *aborts*, or stops, the program if it fails. The `test_code` can return `void`. You don't need to return a value since a problem will halt your program when an `assert` fails.

You'll start with a failing test. Add a single line to the new `test_code` function:

```
void test_code()
{
    assert(0==1);
}
```

When you build and run this, you will see an error something like:

```
./input
input_with_tests.cpp:53: void test_code(): Assertion `0==1' failed.
Aborted
```

The exact message and line number might be different for you, but you can see that an assertion failed. That's a good thing. Now you can delete the `assert(0==1)` line, because you proved that you'll get an error if something is wrong.



The `assert` macro, a primitive way of writing a general function, is often only active in debug mode. If you don't see an error message like `Aborted` or `SIGABRT`, consult the documentation for your compiler flags.

Now that you have a place to write tests, you can go back to the numeric input itself. I'll walk you through adding a test and a new function. Again, you'll write a failing test first then make it pass. I'll do this slowly, and you might spot possible problems or think of questions as you read. Bear with me. By the end of the next section, you will have a small working program to get numeric input, and you'll know more about writing your own functions.

Breaking Your Code into Functions

Your `input_with_tests.cpp` file now has a basic outline of `main` and a test function, but it doesn't do anything much yet. You're going to write a function that gets a number, so call it `get_number`. A function can return `void` or a specific type and can take zero or more parameters. Instead of using `std::cin`, you can use a more general stream type so your test doesn't have to wait for user input. The function needs at least one parameter taking in a stream from which it can extract numbers, but it also needs to report either a number or an error. You can report both in a few ways.

First, you can take a parameter using a reference, indicated by `&`. This is called *passing by reference*. A *reference* refers to something that already exists. The function can then change the original value, defined outside the function. Second, without the ampersand, the parameter is copied. This is called *passing by value*. The function then cannot change the original value, but only a copy of it. You therefore have two ways to pass parameters: either copy the value or allow the function to refer to, and possibly change, it:

```
void a_function(int value); ❶  
void another_function(int & value); ❷
```

- ❶ Passes by value, so the original cannot be changed
- ❷ Passes by reference, so the original can be changed

The `get_number` function can return either the number or the error and take the other one by reference. [Figure 2-2](#) shows both options.

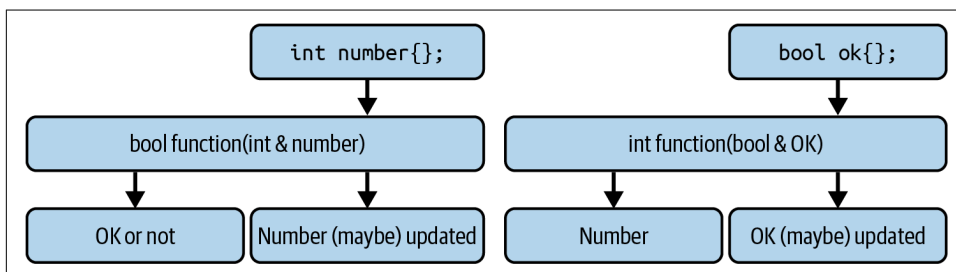


Figure 2-2. You can either return OK and change the number or return some number and change OK

Which way round, though? Let's think this through. If you get a number, you can return that. But what number will you return if there is a problem? If a function promises to return something, it *must* return something. You can return any old number plucked out of the air—let's say 42. If you look back at the function later, though, you might wonder why you chose that value. Instead, you can do something clearer: return a `bool` to indicate an error and take the number by reference.

Let's write a test for a function using this approach.

Starting with a Failing Test, Again

The `get_number` function will get input from a stream and put a number in a `double`, reporting if everything is OK via the returned `bool`. You can use a type of stream called `std::stringstream`, from the `<sstream>` header. This is similar to `std::cin` but allows you to specify characters up front. Using a general stream, rather than reaching out to `std::cin` inside the function, means you can pass in a stream for testing but still use `std::cin` from `main`.

You can put a "1" in a stream, using `{}` to initialize it, like this:

```
std::stringstream some_input{"1"};
```

You can then test your function with this input to make sure you get a 1 back. If so, everything is OK.

Add the code in [Example 2-5](#) to your existing `test_code` function. It won't compile yet, however, because you need to write the `get_number` function.

Example 2-5. A simple test of `get_number` using `assert`

```
#include <sstream> ❶

void test_code() ❷
{
    double value{};
    std::stringstream some_input{"1"}; ❸
    const bool ok = get_number(some_input, value); ❹
    assert(ok); ❺
    assert(value == 1); ❻
}
```

- ❶ Includes the string stream header
- ❷ Adds testing code
- ❸ Uses a string stream instead of `std::cin`

④ Calls a nonexistent function

⑤ Tests ok and value

Sketching out a test has nudged you toward a function declaration, but the code will not compile until you write the `get_number` function.

The return is a `bool`, and you're sending in some kind of general stream and a numeric type. You will read the stream and change the number, so these must be references. That way, they'll allow calling code (the code that invoked the function) to see the changed values. You used a `double` earlier, so do the same now, giving this function a signature (or declaration):

```
bool get_number(some_general_stream & input, double & number);
```

You know `std::cin` is an input stream, so what should you use for `some_general_stream`? The specific input stream type most suitable for ASCII characters is the `std::istream`, which will accept `std::cin` or the `std::stringstream` you just saw. It lives in the `<istream>` header, so add that to your includes.



You can include the headers in any order you like, but people often put them in alphabetical order.

The new function goes in your `input_with_tests.cpp` file, anywhere outside of `main`. For simplicity, write the new function near the top of the file, after your include lines, so the compiler knows about it before you call it in `main` or `test_code`.

Start with the signature you discovered during the tests, and add the simplest thing that makes the code compile, as shown in [Example 2-6](#). The signature says it returns a `bool`, so return a `bool`.

Example 2-6. A function to make the code compile, but leaving the test failing

```
// includes as before
#include <istream> ①

bool get_number(std::istream & input_stream, double & number) ②
{
    return false;
}

// test_code as before ③
// main as before
```

- ❶ Includes the `istream` header
- ❷ Defines your new function
- ❸ Tests and `main` as before

If you build and run this now, your test will fail. The test expects `true`, but the function returns `false`. If you are used to TDD, you will be familiar with this approach.

The test still fails because your function doesn't change the number—it just returns `false`. To make the test pass, you can set the number to 1 and return `true`:

```
bool get_number(std::istream & input_stream, double & number)
{
    number = 1; ❶
    return true; ❷
}
```

- ❶ Sets the number to 1
- ❷ Returns `true`

If you build and run this now, your test will pass. The `get_number` function does just enough for this first test. Let's speed up a little now and add a new test for a failure case after the last test.

Testing Bad Input

Inputting "q" or another letter or two should return `false`. Let's test this too.

Add these extra lines to `test_code`, starting from annotation 1:

```
void test_code()
{
    double value{};
    std::stringstream some_input{"1"};
    const bool ok = get_number(some_input, value);
    assert(ok);
    assert(value == 1);

    double unused{}; ❶
    std::stringstream bad_input{"q"}; ❷
    const bool not_ok = get_number(bad_input, unused); ❸
    assert(!not_ok); ❹
}
```

- ❶ Adds a new test
- ❷ Puts something nonnumeric in a stream
- ❸ Calls the function
- ❹ Checks if things are not OK

Now you have a second test, which fails. That's a good thing. You can add details to the `get_number` function to read the number from the stream and make the test pass.

Use operator `>>` to get input and then the stream's operator `bool` to check that everything is OK. If it is, you can return `true`. Otherwise, return `false`. Add the details to `get_number`, as shown in [Example 2-7](#).

Example 2-7. Function to get a number from a stream

```
bool get_number(std::istream & input_stream, double & number)
{
    input_stream >> number;
    if(input_stream)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Build and run your code, and this time your tests should both pass.



So far, you've put any `if` blocks into curly braces. If you have only one line, though, you don't need to do that. You might therefore see code without the braces:

```
if(input_stream)
    return true;
else
    return false;
```

That's fine, but if you want to add another statement inside the `if` or the `else`, the whitespace might lead you astray. That's why adding braces is sensible, even if they aren't required.

Refactor

Now you have passing tests. In TDD, you start with a failing test, make it pass, and then you *refactor*: take a moment to make the code better or tidier.

You can make this function better and learn a little more C++ on the way. You are capturing the return value in the test, but it's easy to forget to do this. Return values are easily ignored. In [Chapter 3](#) I'll show you a more robust approach to reporting problems using exceptions and another newer C++ feature.

For now, though, you can use a C++17 feature called an *attribute* in the function head to “encourage” the compiler to generate a warning if you ignore the returned value. If you add `[[nodiscard]]` at the start of the function and then discard the return, most compilers will generate a warning. Add `[[nodiscard]]` to your function head and rebuild:

```
[[nodiscard]] bool get_number(std::istream & input_stream, double & number)
```

You can temporarily change the function call in [Example 2-5](#) to ignore the return, and drop the assert for ok:

```
void test_code()
{
    double value{};
    std::stringstream some_input{"1"};
    get_number(some_input, value); ❶
    assert(value == 1); ❷
}
```

❶ Discard the return value

❷ Only one assert left

You will now see a suitable warning:

```
warning: ignoring return value of 'bool get_number(std::istream&, double&)',
declared with attribute 'nodiscard'
```

The exact message might be different, but you will be warned about ignoring a `nodiscard` return value. Useful, right? Put the `test_code` back as it was at the beginning of this section, so you don't discard the return value. Now actually check it:

```
const bool ok = get_number(some_input, value); ❶
assert(ok); ❷
assert(value == 1);
```

❶ Stores the return value

❷ Uses the return value

Your code now compiles, and the tests pass again.

Calling Your New Function from main

You have a function with tests, and you're ready to use it. Call the function from `main`. Add a message asking for a number just before you call your function, shown in [Example 2-8](#).

Example 2-8. Calling tests and a function from main

```
#include <cassert>
#include <istream>
#include <iostream>
#include <sstream>

[[nodiscard]] bool get_number(std::istream & input_stream, double & number)
{
    input_stream >> number;
    if(input_stream)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void test_code()
{
    double value{};
    std::stringstream some_input{"1"};
    const bool ok = get_number(some_input, value);
    assert(ok);
    assert(value == 1);

    double unused{};
    std::stringstream bad_input{"q"};
    const bool not_ok = get_number(bad_input, unused);
    assert(!not_ok);
}

int main()
{
    test_code(); ❶

    double number{};
    std::cout << "Please enter a number.\n"; ❷
    if(get_number(std::cin, number)) ❸
```

```

{
    std::cout << "Got " << number << ", thanks!\n"; ❹
}
else
{
    std::cout << "Something went wrong\n"; ❺
}
}

```

- ❶ Calls the tests
- ❷ Prompts for a number
- ❸ Checks for a number in a function
- ❹ Reports the result if a number is obtained
- ❺ Reports a problem

You now have code to elicit input as well as return output, and you’ve learned how to detect an error on a stream.

I also noted there might be extra input you hadn’t used, so you should learn how to tidy up if that happens. Let’s look at the details.

Understanding Variables, `std::cin`, and Functions in Depth

You learned how to declare a variable, stating the type and name, and how to provide a value using curly braces. You don’t need to provide the value:

```
int number;
```

Since you haven’t assigned `number` a value, it could be anything. It isn’t safe to print this out. Reading an *uninitialized* variable was *undefined behavior* until C++26, which means that anything, including bad things, can happen. C++26 made reading an uninitialized variable *erroneous behavior*, which means you are likely to get a warning or error. Assigning a value to the variable is extra work, and it’s up to you to do this if you need to. C++ can be very efficient, but this efficiency can mean potential problems if you aren’t careful. To avoid trouble, get in the habit of always initializing your variables.

You are also now familiar with `std::cin`. It’s a kind of input stream, or `std::istream`, and has instance member functions you can call, such as `eof` and `fail`. Along with `std::cout`, `std::cin` is an object defined by a *class*. You will learn more about classes later in this book. For now, just know that classes group together functions and can have *member* variables that remember values between function

calls. You have seen how to call instance functions using the dot operator, and static functions using the scope resolution operator.

You checked if everything was OK using `if(input_stream)`, which checks the state of the stream with `fail` and `bad`. Once a stream returns `false`, it will continue to return `false` until you intervene.

Clearing Input Errors

Your code is trying to get a single number. If there's a problem, you could have it try again by adding to the `else` part in `main` in [Example 2-8](#), like this:

```
int main()
{
    test_code();

    double number{};
    std::cout << "Please enter a number.\n>";
    const bool ok = get_number(std::cin, number);
    if(ok)
    {
        std::cout << "Got " << number << ", thanks!\n";
    }
    else
    {
        std::cout << "Something went wrong\n";
        std::cout << "Please enter a number.\n>"; ❶
        const bool ok_now = get_number(std::cin, number); ❷
        if(ok_now) ❸
        {
            std::cout << "Got " << number << ", thanks!\n";
        }
        else
        {
            std::cout << "Something went wrong again\n"; ❹
        }
    }
}
```

- ❶ Prompts for a second attempt
- ❷ Tries to get a number again
- ❸ Checks the stream's state
- ❹ Reports if there is another problem

Try this by typing something nonnumeric, for example a *q*, and then follow that by entering a genuine number on the next attempt. The `get_number` function will continue to return `false`:

```
Please enter a number.  
>q ❶  
Something went wrong  
Please enter a number.  
>1 ❷  
Something went wrong again ❸
```

- ❶ Enters something invalid
- ❷ Tries to enter something valid
- ❸ Shows an error, even for a number

Even though you entered a number on the second attempt, the code still reports a problem.

You can fix this by resetting the stream's state using the `clear` function:

```
input_stream.clear();
```

When you call the `clear` function, the stream will report `true` next time you ask what state it's in. In fact, you *must* call `clear` to use the stream again. (This function can do more—[C++Reference](#)) gives more details if you're interested.)

Now, whatever caused the problem is still in the stream—for example, that *q* you typed. You can tidy that up, too, using the stream's `ignore` function. You tell the `ignore` function how many characters to clear up at most, along with a delimiting character to stop at, such as `\n`. You need to `clear` the stream first and then `ignore` some characters up to a newline character:

```
input_stream.clear(); ❶  
input_stream.ignore( ❷  
    some_length, ❸  
    '\n' ❹  
);
```

- ❶ Clears the error so everything is OK again
- ❷ Ignores leftover characters
- ❸ Until it gets up to some specified length
- ❹ Or until it gets to a newline character

Before you add this to your function in [Example 2-9](#), you need to decide what `some_length` should be.

Now, since you don't know how much is left in the stream after a read, most people tend to pick the maximum possible size when they call `ignore`. What is a stream's maximum possible size? Well, a stream's size is a mysterious type called `std::streamsize`. You can find out the maximum `std::streamsize` by calling `numeric_limits`'s `max` function, which you met earlier.

This time, you'll ask for the `streamsize` version using angle brackets: `std::numeric_limits<std::streamsize>`. As before, the angle brackets `<>` mean you are using a template. Next you'll use the familiar scope resolution operator `::`, followed by the function, `max`, to get the value for `some_length`:

```
std::numeric_limits<std::streamsize>::max()
```

Adding calls to both `clear` and `ignore` in `get_number` will improve problem handling, so do that now. Don't forget to include the `<limits>` header to find the `max` size, as shown in [Example 2-9](#).

Example 2-9. An even better function to get a number from a stream, while handling problems

```
#include <limits>

[[nodiscard]] bool get_number( ❶
    std::istream & input_stream, ❷
    double & number ❸
)
{
    input_stream >> number;
    if(input_stream)
    {
        return true;
    }
    else
    {
        input_stream.clear(); ❹
        input_stream.ignore( ❺
            std::numeric_limits<std::streamsize>::max(), ❻
            '\n' ❻
        );

        return false;
    }
}
```

❶ Provokes a warning if the return is discarded

- ❷ Passes parameters by reference so they can be changed
- ❸ Clears the failed flag
- ❹ Ignores leftover input
- ❺ Until it reaches the maximum possible number of characters
- ❻ Or until it gets to a newline character

You now have a better function to get numeric input. If you left the extra code to call `get_number` again after a failure, you can now enter a *q* and then a number. This time, you won't see "Something went wrong again." Here's a **Godbolt** if you need it. It takes a *q* then a 2, like this:

```
Please enter a number.  
>q  
Something went wrong  
Please enter a number.  
>2  
Got 2, thanks!
```

More on Functions

You've written a couple of functions besides `main`. Let's try some further experiments now to ensure that you understand functions in a bit more detail. I'll suggest changes to make so you can see their effects. You're going to get warnings and errors, and your tests will sometimes fail, but you can put your code back afterward. The best way to learn is often by breaking things. Getting familiar with warnings and errors will also help you learn how to fix things in the future.

Write a new function in your `input_with_tests.cpp` file, somewhere above `main`, and call it from `main`. The `get_number` function takes a `double`. What happens if you use an `int` instead? Let's find out. Try sending an `int` to the `get_number` function, as shown in **Example 2-10**.

Example 2-10. Sending the wrong type to a function (will not compile)

```
void some_experiments()  
{  
    int number{}; ❶  
    bool OK = get_number(std::cin, number);  
    std::cout << OK << '\n';  
}
```

- ❶ Changes `number` to an `int`

You'll get an *error* when you try to compile this code. The exact wording varies between compilers, but you'll see something like one of the following:

```
error: cannot bind non-const lvalue reference of type 'double&'
      to a value of type 'int'
```

or

```
error: no matching function for call to 'get_number'
note: candidate function not viable: no known conversion from
      'int' to 'double &' for 2nd argument
```

or

```
error C2664: 'bool get_number(std::istream &,double &)':
      cannot convert argument 2 from 'int' to 'double &'
input.cpp(55): note: see declaration of 'get_number'
input.cpp(89): note: while trying to match the argument list
      '(std::istream, int)'
```

The compiler sees a function that expects a reference to a double, so it complains when you send in an int. This is a good thing. C++ is a *strongly typed* language, and you will use types to help you write better code over the course of this book. If a function expects a double, you must give it a double. You can use the type system to find errors at compile time, which is another reason C++ is powerful.

Change `number` to a double so the code will compile again. Then let's try one more experiment with this function.

The parameters to `get_number` are passed by reference, using the `&` symbol. Drop the `&` symbol for the second parameter:

```
[[nodiscard]] bool get_number(std::istream &is, double number)
```

Now build your code again, and you won't see any errors or warnings. When you run your program, your tests will fail. Using `assert` isn't as helpful as a proper testing framework, so you'll only see a message similar to this one:

```
Assertion 'value == 1' failed
```

The test code declares a number `double value{};`, which starts with a value of zero. Without the reference symbol `&`, the `get_number` function takes the double by value, so it only sees a copy of the original. Any changes happen to the copy in the function, not to the original value, which remains 0. The `assert` is therefore checking if `value` (which is 0) equals 1, so it fails. Now put the `&` back and rebuild to check that everything is OK again.

Let's think about functions a bit more. You wrote some tests and checked that everything worked:

```

void test_code()
{
    double value{};
    std::stringstream some_input{"1"};
    const bool ok = get_number(some_input, value);
    assert(ok);
    assert(value == 1);

    double unused{};
    std::stringstream bad_input{"q"};
    const bool not_ok = get_number(bad_input, unused);
    assert(!not_ok);
}

```

You cannot see `value` or any of the other variables from outside the function. If you try to use the function's variable `value` from `main`, you'll get an error. Try it:

```

int main()
{
    test_code(); ❶
    assert(value == 1); ❷
    // rest of code as before
}

```

❶ Calls `test_code` where `value` lives

❷ Tries to use `value`

You get an error citing `use of undeclared identifier value`. A function's variables “live” for the *scope* of the function only, and you can't reach inside a function to use its *local* variables. That might be familiar if you're used to other programming languages. Now, remove the `assert` line that broke your code.

Let's try something else using scopes. You can put blocks of code inside curly braces in a function to break it into chunks. This is called *block scope*. Why is it useful? Well, try to use the same variable names for both tests. Either think through what might happen, or try it:

```

void test_code()
{
    double value{};❶
    std::stringstream input{"1"};
    const bool ok = get_number(input, value);
    assert(ok);
    assert(value == 1);

    double value{};❷
    std::stringstream input{"q"};
    const bool ok = get_number(input, value);
    assert(!ok);
}

```


- ❶ Declares a variable called `value`
- ❷ Declares another variable in the same scope, also called `value`

If you try to build the code now, you'll get errors complaining about a redefinition of `value`, `input` and `OK`. You could remove the second declaration, and the error will go, but reusing a variable for a different purpose can confuse people. Instead, if you put each test into a separate block, the variables only live to the end of the block—that is, until the closing `}`. The version in [Example 2-11](#) will therefore compile.

Example 2-11. Tests in smaller block scope

```
void test_code()
{
    { ❶
        double value{}; ❷
        std::stringstream input{"1"};
        const bool OK = get_number(input, value);
        assert(OK);
        assert(value == 1);
    } ❸

    { ❹
        double value{}; ❺
        std::stringstream input{"q"};
        const bool OK = get_number(input, value);
        assert(!OK);
    } ❻
}
```

- ❶ Opens a new scope
- ❷ Declares a variable called `value`
- ❸ Closes the scope, so `value` is *out of scope*
- ❹ Opens another new scope
- ❺ Declares a new variable called `value`, in the new scope
- ❻ Closes the second scope, so the second `value` is also *out of scope*

Splitting the two blocks into separate functions is another option, but this demonstrates a bit more about scope.



Knowing that variables are visible only inside a scope is important. However, if you find you have separate blocks inside a function, this might indicate that you should split your function into smaller, named functions.

When a variable goes out of scope, it no longer exists, which frees up memory. Some languages, like Java and Python, use a *garbage collector* to tidy up, so you can't be sure when this will happen. In contrast, C++ gives you precise control, which can help you keep your programs smaller and faster.

Conclusion

You have seen lots of new C++ in this chapter, though the main aim was learning to get input. You had to declare variables, and you practiced writing functions. You used the stream extraction operator `>>` to get input for an `int` and a `double` in the process.

You needed a variable to hold input, so you learned about declaring and initializing variables:

- Variables have a type, like `int` or `double`. They might not be initialized, which can be problematic.
- You can use a single equal sign, `=`, to assign an initial value. You can also use empty braces, `{}`, or even braces with a value if you want something other than the default.
- You can declare a “variable” as `const`, meaning you can't change its value later.

You now know the basics of declaring variables. You wrote a test and input function. These two functions introduced some important ideas:

- Taking parameters by reference, using `&`, allows you to change their values. This is called passing by reference.
- Without an `&`, the parameters are copied and changed only locally in a function. This is called passing by value.
- The `[[nodiscard]]` attribute provokes a warning if the return value is ignored.
- You must send parameters that match the expected types; otherwise, you'll get an error.

You have used several keywords, parts of the core language, and library functions too.

You used `assert` to write some tests. Templates and classes got a brief mention and will make appearances throughout the rest of this book. You saw how to call class instance functions using the dot operator, and static member functions using the

scope resolution operator. You also caused a few compiler warnings, at my encouragement. Warnings are useful, but they don't stop the code from compiling. They might suggest that you've forgotten something, and you'll see many more of them as you learn C++. Pay attention to them.

You started to think about error handling, returning a `bool` to indicate success or failure. In [Chapter 3](#), you will learn alternative ways to indicate problems, including exception handling.

Exceptions and Expectations

You can now write a short program, and you’ve started to think about potential problems with input. I showed you how to write a function returning a `bool` to indicate success or otherwise, but there are more refined ways of handling problems. This chapter will demonstrate two approaches: exceptions and expectations. You will learn other elements of C++, getting more practice writing functions and building a deeper understanding of block scope.

C++ has supported exceptions for a long time, but some people writing embedded code prefer not to use them because they have an overhead. C++23 introduced a new type called `std::expected`, which holds either a return value or an error. I will show you both approaches in this chapter, and then you’ll be ready to build a larger program in [Chapter 4](#).

Exceptions

Most languages provide a way to raise an exception if a problem happens and provide ways to handle the situation. As with other programming languages, if you try a statement that raises an exception, the program jumps to another location. Being able to jump out of a function or to another place if something goes wrong might seem like magic. The specifics of how exceptions work vary between toolchains, but jumping elsewhere needs some extra housekeeping. You can specify a function as `noexcept` to declare that it will not throw exceptions, which allows the toolchain to avoid some overhead. You’ll see some examples of `noexcept` later in [Chapter 11](#).

If you detect an exception, you also need to decide what to do about it. Sometimes doing nothing and reporting the problem to the outside world is best. In [Chapter 2](#), you wrote a function to get a number and called it from `main`:

```
const bool OK = get_number(std::cin, number);
```

If you got `false`, the code simply said something went wrong. In [Chapter 4](#), you will try to get several numbers, but what will you do if a problem happens? You could stop the entire program, but then the user will have to reinput everything. If the data is coming over the wire or from a file or another process, it might be OK to halt and report an error. Someone can then fix the underlying problem and resend the data. Error handling always throws up different options, whether you return a `bool` or throw an exception or pick a different approach. Deciding what to do usually depends on the context.

Let's stick with a program that inputs a single number in this chapter. In the next chapter you will see how to get and store several numbers. You will use exceptions to indicate problems in this section. In the next section you will learn a different approach.

An *exception* is an event that indicates a problem at runtime, stopping the normal execution or flow of the program. There are three parts to exceptions. First, code can *throw* an exception. You can think of that like throwing a ball, as shown in [Figure 3-1](#). You throw to indicate a problem.

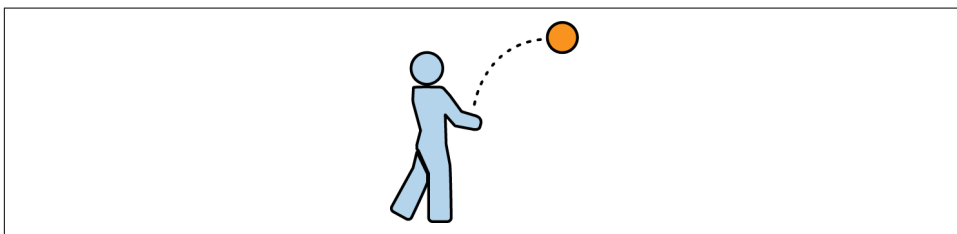


Figure 3-1. Throwing an exception

Second, you may have guessed this, calling code can *catch* an exception. Third, if calling code knows a function might throw, it will *try* to call the code. The *try* and *catch* go together in the calling code. You can attempt to call code in a function without a *try* and *catch* and allow an exception to be caught elsewhere, but you will put the *try* around the potentially throwing code first and dig deeper later. The *throw* happens in the code that is called. [Figure 3-2](#) gives you an overview of the *try* calling a function, and the *throw* jumping back to a *catch* block.

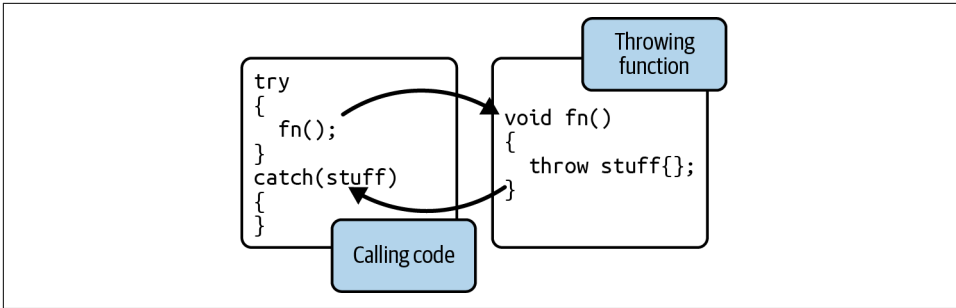


Figure 3-2. Trying, throwing, and catching exceptions

In the previous chapter, you got a number and responded depending on the `bool` that was returned:

```

if(get_number(std::cin, number))
{
    std::cout << "Got " << number << ", thanks!\n";
}
else
{
    std::cout << "Something went wrong\n";
}

```

Let's use exceptions instead of a `bool`. Create a new file called *input_with_exception.cpp*, and write an empty `main` function, which you can probably do on autopilot by now. Look back at an example in [Example 2-8](#) if you need to.

Our last `get_number` had this signature:

```
[[nodiscard]] bool get_number(std::istream & input_stream, double & number);
```

Without the `bool`, you can return the number:

```
double get_number(std::istream & input_stream);
```

You could argue about the `nodiscard` attribute. If someone ignores the number they asked for, that's a bit silly, whereas ignoring the success or failure is arguably more important. If you throw an exception, the calling code cannot ignore the problem. If the exception is not handled, the person running the program will find out about the problem, as you will see.

Throwing Exceptions

Let's raise an exception if the input isn't a number. Based on [Example 2-8](#), write another `get_number` function in your new source file. As before, use a `double` to get the input and check the stream using `if(input_stream)`.

You can now return the number itself if everything is OK and throw an exception if there is a problem. There are many different exceptions, but the most basic lives in the `<exception>` header. You don't even need to throw an exception type. You could throw an `int` or any other type, but doing so is unconventional and will confuse other programmers. [Example 3-1](#) shows the new `get_number` function.

Example 3-1. Throwing an exception if something goes wrong

```
#include <exception> ❶
#include <iostream>

double get_number(std::istream & input_stream) ❷
{
    double number{}; ❸
    input_stream >> number;
    if(input_stream)
    {
        return number; ❹
    }
    throw std::exception{}; ❺
}
```

- ❶ Includes the header for a `std::exception`
- ❷ Declares we'll return a `double` this time
- ❸ Declares and initializes a `double`
- ❹ Returns the number because everything is OK
- ❺ Throws an exception, default initialized with `{}`

Call your new function from `main`:

```
int main()
{
    std::cout << "Please enter a number.\n>";
    double number = get_number(std::cin);
    std::cout << "Got " << number << " thanks!\n";
}
```

Save your file, build your code, and have a play. Try some numbers, and then try some garbage or other characters. If you don't provide a number, your program will terminate with an error message. The exact details vary, but GCC says:

```
terminate called after throwing an instance of 'std::exception'
what():  std::exception
```

Your code used `throw`, but the error tells you more. *Terminate* means the program stopped since it didn't know what to do with the exception. `what` is a function that provides more details, in this case the exception type. The basic `std::exception` you used doesn't give details beyond its type.

Bear in mind I told you there are *three* elements to exceptions. You've seen `throw`, so you are one-third of the way there. I warned you that anyone running a program that throws an unhandled exception will find out about the problem, because they will see errors if something goes wrong.

Trying and Catching

When you experimented, you will have seen some serious-sounding errors if you have nonnumeric input. When internal problems with code leak into public view, the error messages can be off-putting. Sometimes such software failures are called a “Kevlin Henney”. Rather than leaving the scary output if something goes wrong, you can add code at the calling site to handle exceptions.

Handling Exceptions with a try/catch Block

You called your function from `main`:

```
double number = get_number(std::cin);
```

If nothing goes wrong, the next line is executed. If an exception is thrown, the program jumps to the *end of the current scope*, or the next closing brace. The program then hunts for an appropriate *catch block*. If the calling code doesn't catch the exception, the program goes to the code that called the calling code. It may go all the way back to `main` and then give up, showing the error to the user.

The scope of your call to `get_number` is the `main` function, but you can put code inside a smaller block too. You split your test code into small blocks in [Example 2-8](#), and you can do the same for the function call here.

You can try to call a function or evaluate an expression by adding the word `try` at the start of a block surrounding the potentially throwing code. You also need to add a `catch` block to handle exceptions and specify which type of exception you want to catch. If you want to catch more than one type, you can write a `catch` block for each type. We will just deal with one type here and consider further types later.

Example 3-1 throws a `std::exception`. Using the same source file, add *exception handling* code to `main`, as shown in **Example 3-2**. Nothing before `main` needs changing.

Example 3-2. Trying code and catching an exception if something goes wrong

```
int main()
{
    try ❶
    { ❷
        std::cout << "Please enter a number.\n";
        double number = get_number(std::cin);
        std::cout << "Got " << number << " thanks!\n";
    } ❸
    catch(const std::exception & ex) ❹
    { ❺
        std::cout << "Something went wrong\n";
    } ❻
}
```

- ❶ Indicates a try block
- ❷ Opens the block's scope
- ❸ Closes the scope
- ❹ Indicates where to jump to if a `std::exception` happens
- ❺ Opens the scope for the handling code
- ❻ Closes the scope

Look at the catch line:

```
catch(const std::exception & ex)
```

You've seen `const` before. This means you can't change the exception. The `&` indicates *by reference*, so the code doesn't copy the exception. It is common to call an exception `ex`, but you can give it any name you like.

Type the code into your CPP file; then save, build, and run your program. Try some experiments. If you type numbers, the program says "thanks!" after reporting the number. You can get away with something starting with a number, like `1a`, but you know that's left input in the stream after extracting the digits. You saw how to clear extra input in **Example 2-9**.

If you don't enter a number, the `get_number` function throws an exception, and the code jumps to the nearest catch block. This means the line reporting the number and thanking you isn't called. Instead, you will see "Something went wrong". This doesn't tell the user what is wrong, and I am sure you can think of a clearer message. However, it's less scary than `Terminate`.

Expectations

Some people argue that invalid user input isn't really *exceptional*. If you reserve exceptions for truly exceptional circumstances, you can return a `bool` as you did in the previous chapter or use a newer C++ feature, called `std::expected`. You use this type to hold either an *expected* value or an *unexpected* value. Learning to use `std::expected` will teach you a few more C++ ideas, so even if you would rather use exceptions, you will know lots more C++ by the end of this section.



The `expected` type is a new feature introduced in C++23, so older compilers might not support it. If you run into problems, you can try a [Godbolt](#) I made for you.

Create another source file called `input_with_expectation.cpp`, adding an empty `main` function. Include the `<expected>` header at the top, giving you access to the *class template* you need for this section. You met class templates when you learned how to clear input errors in “[Clearing Input Errors](#)” on page 34. You will meet many more over the rest of the book. So, start like this:

```
#include <expected>

int main()
{
}
```

Let's write a new `get_number` function, returning a `std::expected`. `std::expected` takes two types: one for your expected value and one for the unexpected value if things go wrong. You expect, or hope, to get a `double`. For the unexpected type, you can put a message in a `std::string` if you include the `<string>` header. I will show you further details on `std::string` in [Chapter 9](#), but for now, think of them as a type to hold a message. The new `get_number` function therefore needs to return a `std::expected<double, std::string>`. When types are in angle brackets, you know you're using a *template*. If you get a number, you can return the value, since the `std::expected<double, std::string>` can be set to a `double`. If things go wrong, explicitly return `std::unexpected` with a suitable message, as shown in [Example 3-3](#).

Example 3-3. Returning and using a `std::expected`

```
#include <expected> ❶
#include <iostream>
#include <string> ❷

std::expected<double, std::string> get_number(std::istream & input_stream) ❸
{
    double number{};
    input_stream >> number;
    if(input_stream)
    {
        return number; ❹
    }
    return std::unexpected{"That's not a number"}; ❺
}
```

- ❶ Includes the appropriate header for `std::expected`
- ❷ Includes standard strings for a message
- ❸ Starts a function returning the class template
- ❹ Puts the `double` value in the return value
- ❺ Returns a message if something unexpected happens

The `main` function needs some changes too, since it no longer gets a `double` from the function. You check if the `expected` has a value by calling, you guessed it, `has_value`. If it does, the `value` function tells you the value. If it doesn't, you call the error function to find out what happened.

The new version of `get_number` now returns a `std::expected<double, std::string>`, which is a bit of a mouthful. You can simply declare an `auto` (see “Almost always `auto`” on page 51) for the type when you call your new function, rather than stating the type in full. The compiler will fill in the appropriate type for you:

```
int main()
{
    std::cout << "Please enter a number.\n>";
    auto number = get_number(std::cin); ❶
    if(number.has_value()) ❷
    {
        std::cout << "Got " << number.value() << " thanks!\n"; ❸
    }
    else
    {

```

```

        std::cout << number.error() << '\n'; ❹
    }
}

```

- ❶ Gets the return value using `auto` to avoid spelling out the full type
- ❷ Checks if everything is OK
- ❸ Uses the value
- ❹ Reports the unexpected error

Almost always auto

The keyword `auto` became a placeholder type specifier in C++11. The actual type is replaced by the type used to initialize the variable. In the following code, `x` is an `int`:

```
auto x{42};
```

Some people almost always use `auto`. For the `int` the benefit isn't clear. However, for the `std::expected<double, std::string>`, which is more complicated, it's less typing and potentially less error-prone. The `auto` version will be the exact type it is initialized with, so you won't accidentally convert from a slightly different type, for example, if you have a typo in your type.

Being familiar with `std::expected` is useful, though many older codebases will still be using exceptions. You have now seen three different ways to indicate problems:

- Returning a `bool`, which you could extend to return an `int` with a numeric error for any problems in code as the `main` function does
- Throwing exceptions
- Using `std::expected`

I have shown you the basics, but there are a few more details for exceptions and expectations that are worth knowing. Let's dive deeper.

Understanding Exceptions and Expectations in More Depth

You have seen two approaches to handling problems in this chapter. In this section, I will explain each in more detail. Let's start with exceptions, based on the `get_number` function from [Example 3-1](#), using `main` from [Example 3-2](#). Either add to the original file, `input_with_exception.cpp`, or make a copy.

Knowing how to throw and catch exceptions is important. Many standard library functions can throw exceptions, so you are now better prepared to use more standard C++ features. I have only shown you the `std::exception` so far and how to catch one exception. Let's explore some other exception types and see how to catch more than one type.

Other Exception Types

The `std::exception` is one type of exception. CppReference lists [several other types](#), and you can even write your own. You will learn more in [“Adding Another Derived Type” on page 264](#). Let's try another type of exception.

Suppose you want to restrict the range of numbers a user can provide, rejecting negative numbers. The standard library header `<stdexcept>` provides several exception classes, including `std::invalid_argument`, which is suitable for invalid values, negative numbers in our case. You must provide a message when you throw an `invalid_argument`, like this:

```
throw std::invalid_argument("Please provide a nonnegative number");
```

You don't need a new line `\n` at the end of the message. Code catching the exception can add that, or embed the message in longer feedback.

Make a new source file, called *exception_practice.cpp*. You need a `main` function and a `get_number` function as before. This time, extend the `get_number` function to check that the input is non-negative. The `>` symbol means greater than, and `>=` means greater than or equal to. Throw the new exception type if needed, providing a message, as shown in [Example 3-4](#). For now, you can copy the previous `main` function.

Example 3-4. Throwing different exception types

```
#include <exception>
#include <iostream>
#include <stdexcept> ❶

double get_number(std::istream & input_stream)
{
    double number{};
    input_stream >> number;
    if(input_stream)
    {
        if(number >= 0.0) ❷
        {
            return number;
        }
        throw std::invalid_argument("Please provide a nonnegative number"); ❸
    }
}
```

```
    throw std::exception{}; ❹
}
```

- ❶ Includes specific standard exception types
- ❷ Checks if the number is valid
- ❸ Throws an exception with a message
- ❹ Throws a general exception for nonnumeric input



The code compares the `double` `number` to the `double` `0.0`. You could compare the `double` with the `int` `0`, since C++ will automatically convert an `int` to a `double`. This is called an *implicit conversion*, and [C++Reference](#) has further details. Note that some `doubles` don't exactly equal an `int`, so you might get a warning if you try to go from a `double` to an `int`. Herb Sutter and Andrei Alexandrescu wrote a book called *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, where they said implicit conversions can often do more damage than good. It's often better to be explicit.

What happens if you enter a negative number, anything less than zero?

The `catch` block in `main` handles only a general `std::exception`:

```
catch(const std::exception & ex)
{
    std::cout << "Something went wrong\n";
}
```

You will see only the `Something went wrong` message. The `std::invalid_argument` exception is a type of `std::exception`, so the `catch` block can handle it. However, you went to the trouble to add a message when you threw the exception, and that got lost. You can catch the `std::invalid_argument` in a separate `catch` block and do something different there. Specifically, you can call the `std::invalid_argument`'s member function called `what` to get the message.

Let's add another handler for the new type of exception. Because the `std::invalid_argument` is more specific, you need to put that *before* the handler for `std::exception`; otherwise, the more general handler will be found first. Add a new `catch` block to `main`, as shown in [Example 3-5](#).

Example 3-5. Handling different exception types

```
int main()
{
```

```

try
{
    std::cout << "Please enter a number.\n>";
    double number = get_number(std::cin);
    std::cout << "Got " << number << " thanks!\n";
}
catch(const std::invalid_argument & ex) ❶
{
    std::cout << ex.what() << '\n'; ❷
}
catch(const std::exception & ex) ❸
{
    std::cout << "Something went wrong\n";
}
}

```

- ❶ Provides a handler for a more specific exception type
- ❷ Uses the `what` function to find the message, and adds a newline character after the message
- ❸ Catches any other exceptions

Try this version, and some nondigits will still say "Something went wrong", but a negative number now reports "Please provide a nonnegative number".

Position of catch Blocks

You've seen what happens if you throw without a catch block. The problem falls out of `main` and reports terminal messages to the screen. When you put a try/catch block around the call to `get_number`, you can deal with the exception. When an exception is raised, the program *walks the call stack*, jumping back to where it came from and looking for a catch block. If you try to call a function—let's call it `some_fn`—from `main` and that function calls another function, `fn`, the program builds up a call stack, keeping track of where to go back to when a function returns:

```

-> fn()
-> some_fn()
main()

```

If an exception is thrown, the program walks back down the stack, looking for a catch block. This is known as *stack unwinding*. Any variables in blocks between the *throw* and *catch* are tidied up. That means the program might end up further away from the call, as shown in [Figure 3-3](#).

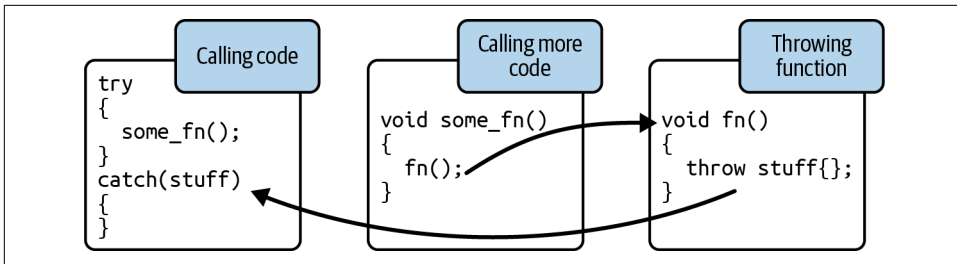


Figure 3-3. Catching exceptions from further away

As you saw, with no catch block on the call stack, the exception is reported to the user. Such an exception is referred to as an *uncaught exception*. The C++ runtime detects this and terminates the program, reporting the error.

Expected Without a Value

Let's now go back to the second approach, using expectations. You made a source file called `input_with_expectation.cpp`, so either add to that or make a copy. Instead of a double, this `get_number` returns a `std::expected<double, std::string>`. Take a look back at [Example 3-3](#) if you need a reminder.

So, what might happen if you don't check `has_value` for a `std::expected`? In [Example 3-3](#) you checked whether the number had a value or not before using it. You can drop that check, and your code will still build:

```
auto number = get_number(std::cin);
std::cout << "Got " << number.value() << " thanks!\n";
```

That may shock some readers, but let's see what happens now if you enter something non-numeric.

GCC gives the following output:

```
Got terminate called after throwing an instance of
'std::bad_expected_access<
    std::__cxx11::basic_string<char,
    std::char_traits<char>,
    std::allocator<char>
>'
>
what(): bad access to std::expected without expected value
```

There are lots of details, but the important part says “bad access to `std::expected` without expected value.”

You've written to `std::cout` before, the character output stream. There's a corresponding `std::cerr`, short for character error stream, which you can use for errors. The specific exception is `std::bad_expected_access`, and you can see the angle

brackets with some type details. You don't need to understand all of these types to get the idea that you tried to access something expected and a bad thing happened!

You can also see the what function giving a slightly less intimidating message. The *signal* is a way for a program to indicate something terminal happened. SIGSEGV means a *segmentation fault* or *violation*. You tried to read something you shouldn't have: the `value`. There are other signals, all starting SIG.

Other toolchains handle problems differently. Visual Studio simply tells me `Abort` was called. You don't need to know all the possible errors by heart, but being used to seeing a problem helps you get a feel for what might have gone wrong.

I presented `std::expected` as an alternative to exceptions, but as you can see, you might have to deal with exceptions if you use it. As long as you check there is a value before you try to obtain it, you are OK. Rather than forcing you to check you have a value first, C++ is allowing you to let the exception be thrown instead. You have a choice.

In fact, you checked `has_value`, but you can use a `std::expected` in a Boolean context, as you did with a stream when you said `if(std::cin)`. Instead of `number.has_value()` you can say `if(number)` in your call:

```
auto number = get_number(std::cin);
if (number) ❶
{
    std::cout << "Got " << number.value() << " thanks!\n";
}
```

❶ Uses a Boolean conversion instead of calling `has_value`

This is shorthand for `has_value`, so either is fine.

There's lots more to `std::expected`, but you now know enough to use this feature.

Conclusion

This chapter has shown you two new ways of indicating and handling problems, allowed you to practice some more C++, and introduced a few new ideas:

- You now know how to try, catch, and throw exceptions.
- You learned more about scope.
- You used another class template, `std::expected`, putting types in angle brackets `<>`.

- You used `auto` rather than spelling out a type in full.
- You initialized a `double` and an exception using curly braces, `{}`.

Now that you're armed with some basic syntax and you can write functions and deal with problems, **Chapter 4** will show you how to build a bigger program and use more of the standard library.

Using Loops, Arrays, and Vectors

You can create code to write output and get input. You got one number so far. In this chapter, I will show you how to store several numbers using *containers*. C++ containers are class templates for common *data structures*, which store elements. C++ has several different types of containers, including *sequenced* containers, like an array or a vector, and *associative* containers, letting you build lookup tables. You will see an example of the latter type in [Chapter 15](#).

In this chapter, you will learn how to fill and use containers. You will also learn how to find properties of the container, such as the largest element. You will learn other useful parts of C++ including loops and more. When I tell you about arrays, I will introduce many new features. By the time you get to the vectors, you will find similar ideas and might be able to guess what to try.

By the end of this chapter you will know a lot more about C++ and be ready to write more detailed programs. In [Chapter 5](#), you will find out how to use various algorithms from the standard library to analyze elements in containers.

Input of Several Numbers Using a Loop

In [Chapter 2](#), you input a single number. Let's think about getting several numbers. How do you get more than one number? You could build on the [initial input code](#), declaring two variables to get two numbers, as shown in [Example 4-1](#).

Example 4-1. Attempt to input several numbers

```
#include <iostream>

int main()
{
```

```

double number1{};
double number2{};
std::cout << "Please enter a number.\n>";
std::cin >> number1;
std::cout << number1 << '\n';
std::cout << "Please enter another number.\n>";
std::cin >> number2;
std::cout << number2 << '\n';
}

```

If you tried this code and entered two numbers, you should see them printed:

```

Please enter a number.
>3 ❶
3
Please enter another number.
>-7 ❷
-7

```

- ❶ You type a number and press Enter.
- ❷ You type another number and press Enter.

The program waits until you type something and press Enter and then shows the number and prompts for a second input.

As you learned in [Chapter 2](#), any nonnumeric input causes an error. The numbers are initialized to 0.0, so you will see zeros output after an error. So, you need some error handling. Furthermore, what should you do if you want three or four or more numbers? The code in [Example 4-1](#) is repetitive, and it is hard to think of good variable names. I hope you agree this is not a sensible approach.

A while Loop

In the previous chapter you wrote [Example 3-3](#) to get one number and handle errors. Let's extend it to get a few numbers. Start a new source file, called *while_input.cpp*. You will copy the `get_number` function from the previous chapter shortly. In [Chapter 5](#), I will show you how to reuse code directly. For now, keep everything in one source file.

C++, like most languages, has ways to *loop*, doing the same thing over and over while a condition is met. You will use a `while` loop, which repeats code while a condition is true. You will meet another kind of loop in [“Displaying and Using the Numbers” on page 68](#).

The value returned from `get_number` is a `std::expected`, and you can use `has_value` to see if it has a value. If the return has a value, you will display the number and get

another value. Otherwise, you stop, jumping out of the loop using the keyword `break`, as shown in [Figure 4-1](#).

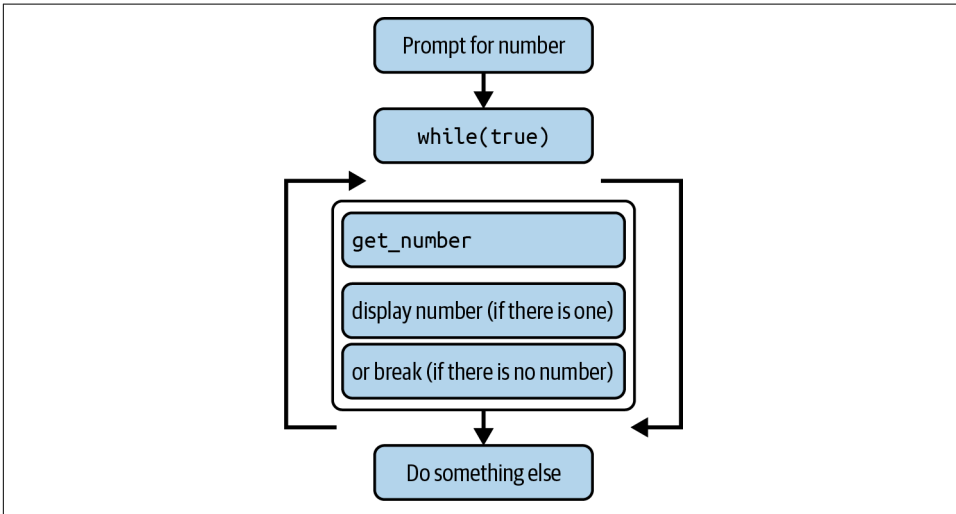


Figure 4-1. Looping while a condition is true

You put the condition in parentheses, as you have done with an `if`. Saying `while(true)` means you need to explicitly break out of the loop when you're done. The code you want to repeat goes in curly braces. You break when you don't get a number, as shown in [Example 4-2](#).

Example 4-2. Getting a number in a loop

```
#include <expected>
#include <iostream>
#include <string>

std::expected<double, std::string> get_number(std::istream & input_stream) ❶
{
    double number{};
    input_stream >> number;
    if(input_stream)
    {
        return number;
    }
    return std::unexpected{"That's not a number"};
}

int main()
{
    std::cout << "Please enter a number.\n>";
    while(true) ❷
```

```

{ ❸
    auto number = get_number(std::cin);
    if(number.has_value()) ❹
    {
        std::cout << "Got " << number.value() << " thanks!\n>";
    }
    else
    {
        std::cout << number.error() << '\n'; ❺
        break; ❻
    }
}
}

```

- ❶ Defines `get_number`
- ❷ Introduces a `while` loop
- ❸ Defines a code block, which can be repeated
- ❹ Checks if the number has a value
- ❺ Displays the error when there is no number
- ❻ Breaks out of the loop

Save, build, and try your program.

You will be able to enter numbers for as long as you want. Your program stops when you try something nonnumeric, like “bye,” for example:

```

Please enter a number.
>4
Got 4 thanks!
>3
Got 3 thanks!
>bye
That's not a number

```

Well done. You can now get lots of numbers. Your code does forget them almost instantly, though. If you want to keep track, you need some more C++ knowledge.

Using an Array

You can use a container from the standard library to hold several elements. There are a few different containers. I will show you the `std::array` first, which is defined in the `<array>` header.

Create a new source file for this section, called `array_input.cpp`. Start with a short function to experiment with arrays. Call it `array_experiment` and include the `<array>` header. Call the function from `main`, ready to add details:

```
#include <array>

void array_experiment()
{
}

int main()
{
    array_experiment();
}
```

An *array* is a class template, so can be used to store (almost) any type. You also specify how many items it will hold. In [Chapter 2](#) you used `std::numeric_limits`, another class template. You can put types in the angle brackets, as you did then (`std::numeric_limits<std::streamsize>`). What you put in the angle brackets are called *template parameters*. You can also use numbers as template parameters. These are referred to as *nontype template parameters*. You are collecting doubles, so tell the `std::array` to use `double` for the first parameter. A `std::array` uses a fixed size as well as a type so has a second nontype template parameter for the size. Let's use five doubles.

You put the type and then the size you require in angle brackets, like this:

```
std::array<double, 5> numbers{};
```

The elements form a *contiguous* block of five doubles, each initialized to `0.0`.

You can provide a few elements, and the compiler can work out what type to use and how many you have, since C++17. This is called *class template argument deduction* (CTAD), because C++ deduces the class template parameters. The following numbers are also a `std::array<double, 5>`:

```
std::array numbers{1.1, 2.2, 3.3, 4.4, 5.5};
```

You access an element using `[]`, specifying the *index* or position you need in the square brackets. The first element is at index 0, so for five elements, the last item is at 4, as shown in [Figure 4-2](#).

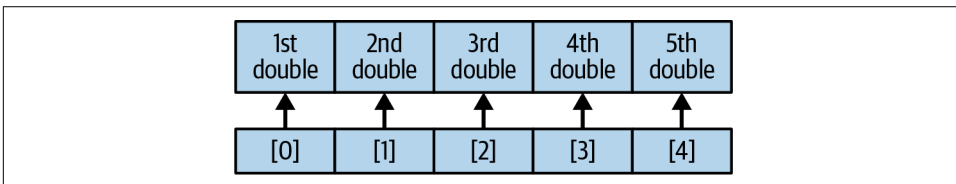


Figure 4-2. Indexing an array of five doubles

Try the code in [Example 4-3](#) in your `array_input.cpp` file. What will output be? Try it if you're not sure.

Example 4-3. Experiment with `std::array`

```
#include <array>
#include <iostream>

void array_experiment()
{
    std::array<double, 5> numbers{}; ❶
    std::cout << numbers[0] << '\n'; ❷
    numbers[0] = 2.5; ❸
    std::cout << numbers[0] << '\n'; ❹
}
```

- ❶ Declares an array of five doubles
- ❷ Displays the first value
- ❸ Changes the first value
- ❹ Displays the updated value

If you run your program, you will see the original value, 0, followed by the updated value, 2.5:

```
0
2.5
```

By default, the output shows a 0.0 as the value 0. You'll see how to control format in [Chapter 9](#).

The numbers all start at zero, and you used the `[0]` to get the first value. You then *assigned* the new value of 2.5 to the first element, using its *index* zero, so see the updated value in the next output.

OK, you are trying to get lots of numbers. You can write a function based on the `get_number` function from [Example 4-2](#). Previously, you stopped when you got non-numeric input. If something goes wrong now, you need to clear problems and ignore nonnumeric input; otherwise, your program will get stuck on problem input. You saw how to tidy up using `clear` and `ignore` in [Chapter 2](#). Let's pull the code together.

Write this version of `get_number` above `main` in your `array_input.cpp` file, using [Example 4-4](#).

Example 4-4. Function to get a number, clearing up if there is a problem

```
#include <expected>
#include <iostream>
#include <limits>

std::expected<double, std::string> get_number(std::istream & input_stream)
{
    double number{};
    input_stream >> number;
    if(input_stream)
    {
        return number;
    }
    input_stream.clear(); ❶
    input_stream.ignore( ❷
        std::numeric_limits<std::streamsize>::max(),
        '\n'
    );
    return std::unexpected{"That's not a number"};
}
```

❶ Clears errors

❷ Mops up unused input

In [Example 4-2](#), you got numbers in a while loop but didn't save them. You can do something similar here, storing the numbers in a `std::array`. The overall code, without details, is a while loop getting numbers, as shown in [Figure 4-3](#).

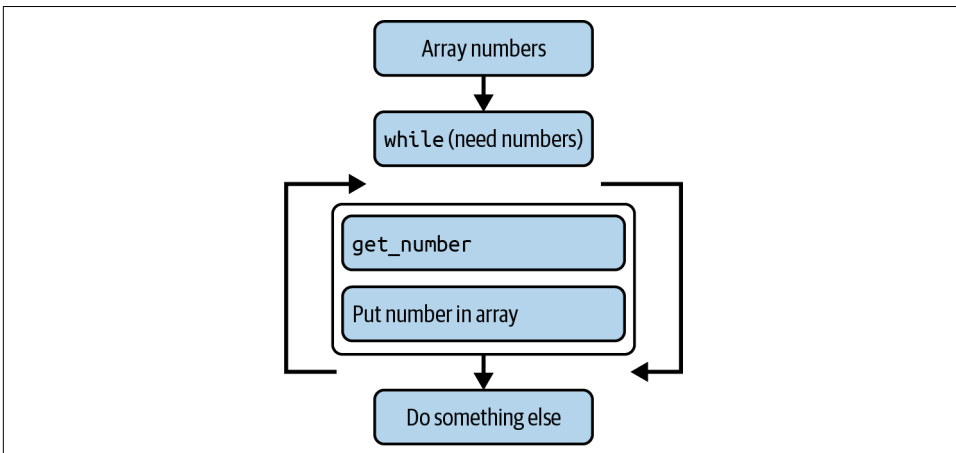


Figure 4-3. Filling an array of five doubles

You need to learn more C++ fundamentals to get this working. I'll talk you through these, and then you can try the new ideas in your `main` function.

You only have a fixed amount of space in the `std::array` you are using. You know your `std::array` has size 5, but you can call its `size` function to find this out, rather than try to remember. If you count how many numbers you get, you can stop before you run out of space in the `std::array`. The count should match the type returned by the `size` function.

You have met the numeric types `int` and `double` so far. These support negative numbers, but a `std::array`, along with other containers, uses an unsigned number for its size. Unsigned numbers include zero and positive numbers and can be represented with a `size_t`. You can add a single `u` to the end when you declare such a value to be precise, saying like this: `size_t count = 0u;`

You can compare `count` with the array's `size` in a `while` loop, seeing if you have fewer numbers than needed. You want to keep looping if `count` is less than (`<`) the `size`:

```
while(count < numbers.size())
```

Now you need to know how to put numbers in the array.

You use operator `[]` to get or set an array element, using a position or index. Look back at [Figure 4-2](#) for a visual reminder. In code, you put the position or index in the square brackets to access an element:

```
numbers[count] = 508; ❶  
double some_other_number = numbers[count]; ❷
```

❶ Sets an element

❷ Gets an element

Lastly, you need to increase the `count`, ready for the next time around the loop. If you don't change `count`, you will overwrite the value at that position in `numbers`. A succinct way to add one to a number uses the increment operator, `++`.

Increment operators

You can add one to a value using `count = count + 1`, but you can also say `++count` or `count++`. Putting the plus signs first is the *preincrement* operator. Consider a value:

```
int x = 0;
```

If you use preincrement on `x` and set a new value, `y` to `x`, like this:

```
int y = ++x;
```

`y` is set to 1, because `x` is incremented first. Both `x` and `y` become 1.

Putting the sign second is the *postincrement* operator. If you use postincrement like this:

```
int y = x++;
```

y is set to 0, the current value of x, first, and then x is incremented. So, x is 1 and y is 0.

There are a few cases where using preincrement rather than postincrement makes a difference, but I will avoid them in this book. They can be difficult to reason about.

You now know four new C++ features you need to get and store several numbers:

- 0u is an *unsigned* whole number.
- You can compare an unsigned number with an array's size, for example, using less than <.
- You use operator [] to access a specific element in an array.
- You can add 1 to a number in various ways, but using ++ to preincrement is common.

Using these new concepts means you can now call your `get_number` function in a loop in the `main` function in `array_input.cpp`, as shown in [Example 4-5](#).

Example 4-5. Main code to get up to five numbers and remember them

```
int main()
{
    std::cout << "Please enter some numbers.\n";
    std::array<double, 5u> numbers{}; ❶
    size_t count{0u}; ❷
    while(count < numbers.size()) ❸
    {
        std::cout << '>';
        auto number = get_number(std::cin);
        if(number.has_value())
        {
            numbers[count] = number.value(); ❹
            std::cout << "Got " << number.value() << " thanks!\n";
        }
        else
        {
            std::cout << number.error() << "\n";
        }
        ++count; ❺
    }
}
```

- ❶ Declares an array of five doubles
- ❷ Starts a count at zero
- ❸ Loops while you haven't entered five things
- ❹ Stores a number at position count
- ❺ Increases count by one

Save and build your code, and try it. You can get away with some nonnumeric input as well as numbers:

```
Please enter some numbers.  
>1  
Got 1 thanks!  
>meh  
That's not a number  
>3  
Got 3 thanks!  
>4  
Got 4 thanks!  
>-889900  
Got -889900 thanks!
```

You have gone to the trouble of storing the numbers, but you're not yet doing anything with them. Let's use the array and learn more C++.

Displaying and Using the Numbers

Let's start with a way to display the values. C++ provides a special type of loop for containers, called a *range-based for loop*, which lets you access the elements in sequence.

Let's use `std::cout` to display the numbers. The range-based for loop starts with the word `for`, followed by parentheses. Inside the parentheses you choose a name of each element, like `number`, and then add a colon and the container's name. You can make the elements `const` since you won't change them, and to save remembering what the element types are, you can use `auto`, which you met in [Chapter 3](#). The body of the for loop goes inside curly braces, like the `while` loop. You can print the current element here, so your program displays something.

Try the code, shown in [Example 4-6](#).

Example 4-6. Populating and displaying an array

```
void show_numbers(const std::array<double, 5u> & numbers)
{
    for(const auto number: numbers) ❶
    {
        std::cout << number << '\n'; ❷
    }
}
```

- ❶ Declares a range-based for loop over the numbers
- ❷ Displays the current number

You can call this in `main`, near the bottom after you get the input:

```
int main()
{
    // ... ❶
    show_numbers(numbers); ❷
}
```

- ❶ Code as before
- ❷ Calls the new function to display numbers

Try your program again. If you give some nonnumeric input, you will see a zero:

```
Please enter some numbers.
>3
Got 3 thanks!
>4
Got 4 thanks!
>f
That's not a number
>6
Got 6 thanks!
>7
Got 7 thanks!
3
4
0 ❶
6
7
```

- ❶ The third entry wasn't a number, so the array element there remains at zero.

You can do various things with your container of numbers and a range-based for loop. Let's find the largest number. If you start with the first number, `numbers[0]`, you

can compare this with the other values, updating the biggest if needed, as shown in [Example 4-7](#).

Example 4-7. Find the biggest number

```
void max_number(const std::array<double, 5u> & numbers)
{
    double biggest = numbers[0]; ❶
    for(const auto number: numbers) ❷
    {
        if(number > biggest) ❸
        {
            biggest = number; ❹
        }
    }
    std::cout << "The biggest number is " << biggest << '\n'; ❺
}
```

- ❶ Stores the first number
- ❷ Range-based for loop
- ❸ Checks if the current number is bigger
- ❹ Updates the biggest if it is
- ❺ Displays the biggest

Call this from `main` and try it:

```
int main()
{
    // ...
    show_numbers(numbers); ❶
    max_number(numbers); ❷
}
```

- ❶ Code as before
- ❷ Calls the new function to find the biggest numbers

You will see the biggest number displayed. If you are wondering if you can avoid finding the first number and then looking at it again in a loop, you are getting ahead of me. The standard library actually has some algorithms you can use instead to find maximums, minimums, and more. I will show you some of these in [Chapter 5](#).

You have covered a lot of ground so far. Well done!

The `std::array` needs a size in advance, but you don't always know how many elements are needed up front. C++ provides another container, called a `std::vector`, which grows on demand. The `std::vector` is a sensible container to reach for first when you code, because it is more flexible than the `std::array`. I showed you the `std::array` first because it is simpler. Some of the `std::vector` functions, like accessing an element, are exactly like the `std::array`, so you will revise a bit while you learn even more. Let's use the `std::vector` next.

Using a Vector

Create a new source file called *vector_input.cpp* for this section. The `std::vector` lives in the `<vector>` header, so include that and write an empty `main` function.

The `std::vector` is another class template, like `std::array`. This time, you only specify the elements' type. You don't fix the size because elements can be added or removed. Like the `std::array`, a `std::vector` has a `size` function, telling you how many elements it contains. You can state the type in angle brackets, for example:

```
std::vector<int> numbers{};
```

`numbers` then starts with size 0. You can also provide elements in the curly braces instead:

```
std::vector<int> numbers{0, 1};
```

`numbers` then starts with size 2 and contains a 0 and a 1. As you saw for `std::array`, the compiler can figure out the type when you provide some elements, so you don't always need to specify the type:

```
std::vector numbers{0, 1};
```

You saw how to display all the elements in a `std::array` in [Example 4-6](#) and can use another range-based for loop to display the vector. Try the code shown in [Example 4-8](#).

Example 4-8. Populating and displaying a vector

```
#include <iostream> ❶
#include <vector>    ❷

void vector_experiment()
{
    std::vector numbers{0, 1}; ❸
    for(const auto number: numbers) ❹
    {
        std::cout << number << '\n'; ❺
    }
}
```



```
int main()
{
    vector_experiment();
}
```

- ❶ Includes `iostream` for output
- ❷ Includes the `vector` class template
- ❸ Puts 0 and 1 in a vector
- ❹ Declares a range-based for loop over the numbers
- ❺ Displays the current number

You could swap the `std::vector` in [Example 4-8](#) to a `std::array` and get the same output. All the containers have a common subset of functions. This allows C++ to provide useful library features that work for any container. I will show you some of these in [Chapter 5](#). Each container has different extra features, though, and behave in different ways. I'll tell you more in “[Understanding Sequential Containers in More Depth](#)” on [page 76](#). Let's take a look at some `std::vector` specialties.

Adding More Elements to a Vector

You can't add more elements to an array. You can change an array's elements, but never its size. In contrast, you can add elements to a vector. Both the array and vector's elements have a beginning and an end, and the elements are stored contiguously. The exact location of a `std::vector`'s elements might change as you add or remove elements.

You don't need to know all these details to start using a `std::vector`. However, this does mean a `std::vector` needs housekeeping, unlike a `std::array`, as illustrated in [Figure 4-4](#).

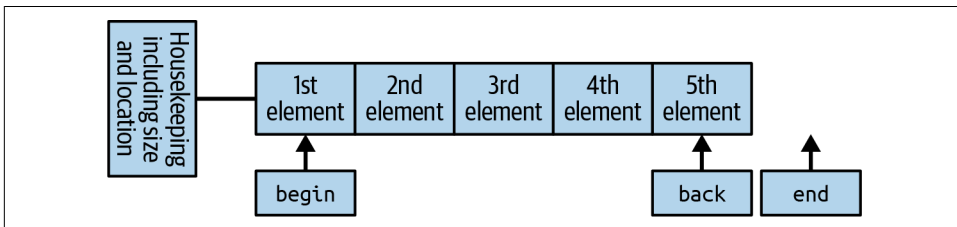


Figure 4-4. A vector with some elements

You can insert elements anywhere in a vector, or you can add a new element at the end. To add to the end of a `std::vector`, you can use the `push_back` function.

Try adding a new number before you print out the numbers in your `vector_experiment` function:

```
void vector_experiment()
{
    std::vector<int> numbers{0, 1};
    numbers.push_back(-123); ❶
    for(const auto number: numbers)
    {
        std::cout << number << '\n';
    }
}
```

❶ Add another element to the end.

Try your code. You will see the numbers printed on separate lines:

```
0
1
-123
```

You can also add elements in between existing items, specifying where and what. [Figure 4-4](#) showed the begin and end of the vector. All containers provide corresponding begin and end functions, which return an *iterator*. There are various types of iterators, and I will show you more details later in [“Using Iterators in Algorithms” on page 94](#).

Let’s write a new function called `vector_insert` and insert a number at the beginning of a `std::vector`. Insert takes a position, so use `begin` for the beginning, as shown in [Example 4-9](#).

Example 4-9. Inserting an item at the beginning of a vector

```
void vector_insert()
{
    std::vector<int> numbers{0, 1};
    numbers.insert(numbers.begin(), -123); ❶
    for(const auto number: numbers)
    {
        std::cout << number << '\n';
    }
}
```

❶ Inserts -123 at the beginning of a vector

Call this from `main` and try your code again. You will see `-123`, `0`, and `1` printed on separate lines.

You can use `end` instead, and the new number will be inserted at the end, as you saw with `push_back`. If you want to use a different position, you can increment the `begin` to move to the second element. You do that with the `++` operator. Look back at “[Increment operators](#)” on page 66 if you need to. Try using this to change the middle value, as shown in [Example 4-10](#).

Example 4-10. Inserting an item into the middle of a vector

```
void vector_insert()
{
    std::vector<int> numbers{0, 1};
    auto iterator = numbers.begin(); ❶
    numbers.insert(++iterator, -123); ❷
    for(const auto number: numbers)
    {
        std::cout << number << '\n';
    }
}

int main()
{
    vector_insert();
}
```

❶ Finds the beginning of a vector

❷ Inserts `-123` in the second position, one after `begin`

You can do much more with a `std::vector`, and you will over the rest of this book.

I’ll show you a few extra features of a vector now, before we return to our original plan to fill a container of numbers.

A Few Other Container Functions

You can jump forward by adding to an iterator:

```
iterator = numbers.begin() + 2;
```

This allows you to move to any element. You can also subtract to go back. You can do this for containers with contiguous elements.

For the `std::array`, you access an element using operator `[]`, specifying the position you need. The `std::vector` lets you do the same. Again, `0` corresponds to the beginning, as you saw in [Figure 4-2](#).



Nothing stops you from moving too far forward or backward with iterators or using an invalid index in operator []. If you end up outside the elements, you have undefined behavior, and bad things might happen. You met undefined behavior in “[Declaring Variables](#)” on page 15. Be careful!

You can use the `at` function instead of using operator [], which checks the position you ask for. If you want to set the 10th element, you can use either function:

```
numbers[9] = 404;
numbers.at(9) = 808;
```

The `at` function will throw an exception if you step outside the elements. What should you do with such an exception, though? If you calculated the index and got it wrong, you have a mistake in your code you need to fix. People therefore tend to use operator [] instead.

Let’s finish off this section by returning to our original problem, getting several numbers in a container.

Getting Several Numbers in a Vector

You have covered a lot of ground in this chapter. You can put some numbers in a vector using the function from [Example 4-4](#), so you have half of what you need already. Copy that function into your `vector_input.cpp` file, above `main`.

You can now delete or comment out your experiment calls in `main` and use similar code to [Example 4-5](#) to get the numbers. This time, you will `push_back` values until the input isn’t a number, as shown in [Example 4-11](#).

Example 4-11. Getting numbers in a vector

```
#include <expected>
#include <iostream>
#include <limits>
#include <vector>

std::expected<double, std::string> get_number(std::istream & input_stream)
{
    double number{};
    input_stream >> number;
    if(input_stream)
    {
        return number;
    }
    input_stream.clear();
    input_stream.ignore(
        std::numeric_limits<std::streamsize>::max(),
```

```

        '\n'
    );
    return std::unexpected{"That's not a number"};
}

int main()
{
    //vector_insert();
    std::cout << "Please enter some numbers.\n>";
    std::vector<double> numbers{};
    auto number = get_number(std::cin);
    while(number.has_value()) ❶
    {
        numbers.push_back(number.value()); ❷
        std::cout << "Got " << number.value() << " thanks!\n>";
        number = get_number(std::cin); ❸
    }
    std::cout << number.error() << '\n';

    std::cout << "You entered\n";
    for(const auto & number: numbers) ❹
    {
        std::cout << number << '\n';
    }
}

```

- ❶ Loops while you have a value
- ❷ Pushes back the value
- ❸ Tries to get another number
- ❹ Displays the numbers

Try your code. You can add lots of numbers now. You will find properties of a container of numbers in [Chapter 5](#), reusing much of this code.

Understanding Sequential Containers in More Depth

You have seen how to use a `std::array` and `std::vector`. They have many similarities. Let's look at initializing either in more depth and then consider a vector in more detail.

Initializing Containers with an Initializer List

You declare a vector and an array in a similar way. You can state the type they contain, but the array needs to know how many elements in advance:

```
std::array<int, 3> numbers;  
std::vector<int> more_numbers;
```

You can also use class template argument deduction (CTAD), so the compiler deduces the template details for you:

```
std::array numbers{1, 4, -3};  
std::vector other_numbers{2, 5, -2};
```

The elements in curly braces used to initialize the containers are called an *initializer list*.

If you try to put different types in such a list, you get an error:

```
std::vector numbers{0, 1, 2.5};
```

The exact words will vary between compilers, but GCC says:

```
error: narrowing conversion of '2.5e+0' from 'double' to 'int'
```

You already know `int` and `double` are different types. An `int` will (usually) fit in a `double`. If you tried to fit a `double` into an `int`, you would lose a decimal part, and the biggest `double` is bigger than an `int`, so you might lose more than a fraction:

```
double x = 0; ❶  
int y = 0.0; ❷
```

- ❶ OK, converting an `int` to a `double` is safe
- ❷ Converting from `double` to `int`, possible loss of data

All the types in the initializer list must match. That's a good thing.

How do you create a vector of doubles? You can either explicitly state you want doubles, using the angle brackets, `std::vector<double> numbers{0, 1, 2.5};`, or make all the numbers the same type, saying `std::vector numbers{0.0, 1.0, 2.5};`.

What Happens When You Add to a Vector

You used `push_back` and `insert` to add elements to a vector. The vector has space for a few items but might run out eventually. When that happens, the `std::vector` *allocates* more space elsewhere, copies the existing values, and then adds your new value. The vector has a `capacity` function showing how many items it has space for. You can ask this before and after a `push_back` call to see what happens:

```
#include <iostream>  
#include <vector>  
  
int main()  
{
```

```

std::vector numbers{0, 1}; ❶
std::cout << "Space for " << numbers.capacity() << '\n'; ❷
numbers.push_back(2); ❸
std::cout << "Space for " << numbers.capacity() << '\n'; ❹
}

```

- ❶ Some numbers in a vector
- ❷ Displays the capacity
- ❸ Pushes a new value to the end
- ❹ Displays the capacity after the new element is added

When I run this, I get a capacity of 2 and then 4. C++ doesn't dictate how many items a vector has capacity for initially, nor how many more are added when needed, so you might get different values. A vector often doubles in size, though.

To keep the elements next to each other, the vector will tidy up the old elements, moving values to a new position and adding extra capacity, as shown in [Figure 4-5](#).

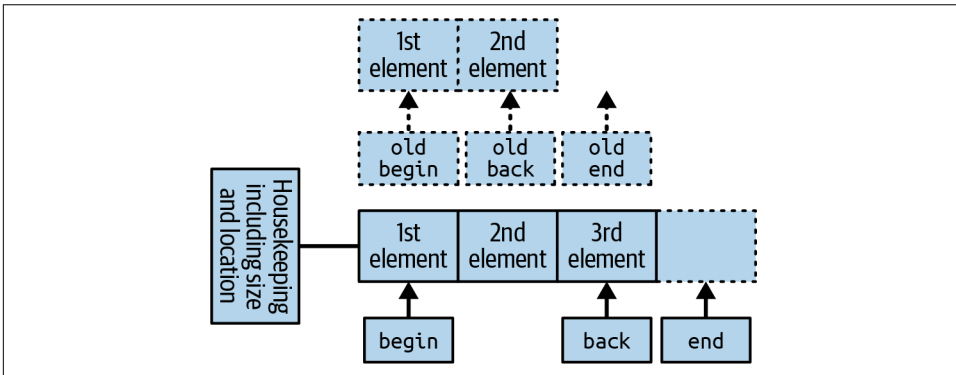


Figure 4-5. Adding elements to a vector might move them all

Keeping the elements in a contiguous block can make accessing them relatively quick. The machine will try to predict where to get data from. When data is contiguous, the machine can predict where to look next, speeding things up.

You can add items to a vector, and you can erase them too.

What Happens When You Delete from a Vector

To remove items from a vector, you use `erase`. You can erase a single item, using an iterator, or erase several items, using a `begin` iterator and one beyond the last item.

Try the following function, calling it from `main`:

```

void remove_from_vector()
{
    std::vector numbers{ 1, 3, 7, 9, 0 }; ❶
    numbers.erase(numbers.begin()); ❷
    numbers.erase(numbers.begin() + 1, numbers.begin() + 2); ❸
    for (const auto number : numbers)
    {
        std::cout << number << '\n';
    }
}

```

- ❶ Declares some numbers in a vector
- ❷ Erases the first number
- ❸ Erases two more numbers

You start with five items and delete the first, leaving 3, 7, 9, 0. You then erase the second up to, but not including, the third (`begin() + 2`), leaving 3, 9, 0. You can insert items too. Both `erase` and `insert` provide several overloads to support these differing requirements.

The vector adds space when needed. If you erase elements, you leave unused space; [Figure 4-5](#) showed extra capacity when new items were inserted. If you check your vectors' capacity, you will see it has space for five elements after you erased elements. That's not a disaster, but the vector does provide a function to reclaim this, called `shrink_to_fit`. If you call that, you will see the capacity drops back to just three:

```

std::cout << "Capacity " << numbers.capacity() << '\n'; ❶
numbers.shrink_to_fit(); ❷
std::cout << "Capacity after a shrink " << numbers.capacity() << '\n'; ❸

```

- ❶ Shows a capacity of 5
- ❷ Shrinks the vector to fit the number of elements
- ❸ Capacity shrinks to 3

Let's consider one more feature of a vector before finishing up.

Initializing a Vector with a Fixed Value

You can initialize an array and vector in the same way, but the vector supports some other approaches.

You can ask for a count elements with a specific value, like this:

```

std::vector<int> numbers(2, 5);

```


The numbers start with two 5s. Notice I used parentheses, `()`, that time. So far you have used curly braces, `{}`, providing an initializer list, like this:

```
std::vector<int> numbers{2, 5};
```

This `std::vector` contains a 2 and a 5.

The curly braces and parentheses are doing something completely different. The curly braces tend to be used to provide a specific value or set of values, while parentheses are almost always doing something else.

Other Sequential Containers

The vector provides `push_back`, allowing you to put an element at the end. There is no `push_front` function, but you did use `insert` to add elements at the start. The lack of `push_front` is a hint that such a function is inappropriate, or at least inefficient, for a vector. After an insert at the front, the vector needs to copy all the existing elements, which might take a while for a large vector.

There are other containers, including a `std::deque`, pronounced *deck* by many people. This container is a *double-ended queue*. The elements are typically stored in fixed-sized arrays, so adding elements at the front and back are relatively efficient. Unlike a vector, whereas `insert` needs to shunt up the subsequent elements, a deque can add a new array at the start, using a new array. This means the deque needs more housekeeping, and iterating through the elements involves jumping to different blocks from time to time.

There are **many other C++ containers**. Each container is designed to support certain operations efficiently, often meaning some operations are slower too. The presence or absence of member functions provides clues about what is possible or sensible.

Conclusion

You met a while loop and the range-based for loop, along with several useful C++ features:

- `size_t` are *unsigned* whole numbers.
- You can use a trailing `u` to specify an unsigned whole number, such as `0u`.
- Less than is `<`, and greater than is `>`.

- You can add 1 to a number in various ways, but using ++ to preincrement is common.
- You can convert an `int` to a `double`, but going the other way will generate a warning.

You also used `std::array` and `std::vector`, two sequential containers from the standard library:

- These are class templates, both using a type, but the array has a fixed size too.
- You can use an initializer list, like `{1, 3, 2}`, to initialize either container.
- You can specify a count and value to initialize a `vector`.
- You use operator `[]` to access a specific element in an array or vector.
- All containers have `begin` and `end` functions.
- You can change the value of elements in an array but cannot change the array size.
- You call `push_back` on a vector to add elements to the end, and this might have to allocate new space and copy the elements.
- You can call `insert` to add elements elsewhere in a vector, which might have to allocate new space and copy the elements.
- A `std::deque` supports `push_front` because the elements are laid out differently from a vector, making this operation quicker.

You have met lots of new C++ in this chapter. If you get used to using `std::vector` and `std::array`, you will be able to do a lot of C++. I recommend coming back and trying the other sequential containers another time.

In [Chapter 5](#), you will use `std::vector` more and see how to use some algorithms from the standard library.

Using Standard Library Algorithms

In [Chapter 4](#), you got several numbers through input and found the largest value in an array. In this chapter, you will do the same thing, this time using some standard library functions. I will also show you how to organize your code sensibly, so you can reuse your functions in different programs.

Let's give the numbers a meaning: stock prices. Armed with a set of stock prices, you can find the largest, the smallest, and other properties. Over the rest of this book, I will show you how to simulate prices, read them from a file, and build a small trading-simulation game.

Getting Several Numbers Into a Vector (Again)

In the previous chapter, you put numbers into an array and then into a vector. The vector allows you to store a varying number of values, so it tends to be most people's default choice of sequential container.

You're going to use your code from [Chapter 4](#) but reorganize it to make it easy to reuse. Each source file you have written so far has its own `main` function. Now, an application can have only one `main` but can include several source files. In this section, you will write two source files with extension `.cpp`, and your own header file, with extension `.h`. One source file will have the usual `main` function and will use the code from the second source file, which you will also reuse in future chapters.



It is a best practice to use the `.cpp` extension for your source files. You don't have to, and some people use `.cc` or `.cxx` or something else instead. However, consistency makes your code's structure clearer. The header extension doesn't matter either, and people often use `.hpp` instead of `.h`. I will use `.cpp` for source files and `.h` for headers, but you might see other conventions elsewhere. As you have seen, the standard libraries' headers don't have an extension—so when there's no extension, you know you are using the standard library.

If you put declarations in header files, you can use them in other source files that include your header. You *could* declare the functions and objects you want to use in the source file directly. However, you might also need to use the same declaration in each of your source files. Adding the same code to each source file isn't very sensible: repetitive code is tedious and error-prone. Using a header file is a much better approach.

When you include a header file, the code from that file gets copied in place of the `#include` line, so you don't need to copy and paste code. Think of the `#include` as saying “copy the contents of the header here,” as shown in [Figure 5-1](#).

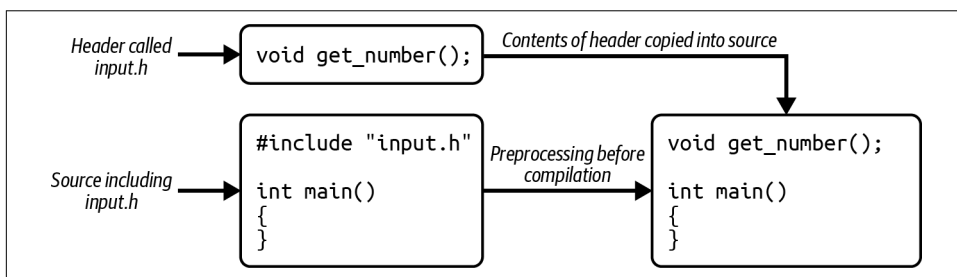


Figure 5-1. Including a header

Start with a new file and call it `input.h`. You will declare the `get_number` function you wrote before in this file. Look back at [Example 4-11](#), and take note of the function head `get_number`. This is the signature for the function. If you use just this part and end it with a semicolon, you are *declaring* a function: saying there's a matching definition somewhere. Copy the following declaration into your header file so you can use it from your main source file:

```
std::expected<double, std::string> get_number(std::istream & input_stream);
```

The function's definition will go in a source file, so you can reuse the header and source files (and you'll need to do so in future chapters).

Your header needs a little more than just the function declaration, though. Let's think through what else is needed.

First, any given header file might be included by several source files or header files. In fact, you could even include your header file *in the header file itself*. That would be a silly thing to do, but if you did, what would happen? Imagine drawing a version of [Figure 5-1](#) where the header file includes itself! The compiler would keep seeing a line asking for the header to be included, open the header, see the same include line, and so on. Fortunately, your header can indicate that it should be included only once. There are a few ways to do this, but writing `#pragma once` at the top of the file works.

Compiler-specific instructions via the pragma directive

Adding a `pragma` statement, sometimes called a *pragma directive*, allows you to use compiler-specific features—though this particular one, `#pragma once`, works on almost all toolchains. Like the `include` statement, the `pragma` directive line starts with a `#`. These are two of several directives used before compilation, which are collectively known as *preprocessing*. For example, you saw how `#include` statements copy in the contents of each included file before compilation. You will also see people using `#ifndef` instead of `#pragma once` in a header, like this:

```
#ifndef A_UNIQUE_NAME ❶
#define A_UNIQUE_NAME ❷

// ... contents of header

#endif ❸
```

❶ Checks `A_UNIQUE_NAME` has not been defined using `#ifndef` (not) `defined`

❷ If not, defines it (as a symbol only)

❸ Ends the `#ifndef`

If the `#ifndef` is true, the unique symbol is then defined on the next line, so the header won't be included a second time. The `#ifndef` ends with a corresponding `#endif`.

So your header file needs `#pragma once` as its first line.

Next, your declaration of `get_number` will use `std::expected`, `std::string`, and `std::istream`. Include those headers too, so the declaration makes sense and the code (including the header) can find definitions or declarations for the types used. If you include just your header but not the includes, your code will error, saying it doesn't know what `std::expected` and so on are. Finally, you will declare your function, `get_number`.

Now, you met namespaces in the first chapter. You've also seen that every time you use something from the standard library, you add `std::` at the start. Putting your functions inside your own namespace is sensible, especially if you want to reuse the code in other programs. Namespaces group related code together and give your code structure. They also restrict scope, which lets you have two functions with the same signature in different places. So let's put your function in a `stock_prices` namespace.

Pulling this together gives you the header file shown in [Example 5-1](#).

Example 5-1. Your first header file, `input.h`

```
#pragma once ❶

#include <expected> ❷
#include <istream> ❷

namespace stock_prices ❸
{
    std::expected<double, std::string> get_number(std::istream & input_stream); ❹
}
```

- ❶ Guards against multiple inclusions
- ❷ Includes `std::expected`, `std::istream`, and `std::string`, used in the function declaration
- ❸ Opens a namespace
- ❹ Declares a function inside the namespace

You have declared a function. Now you need to define it. Create a new source file called `input.cpp` for the definition. (People usually give header files the same name as the source file, with a different extension. If you see code that includes `input.h`, you can expect the definitions to be in a file called `input.cpp`.)

The `get_number` definition will go in your new source file, but first, there are a couple of things you need to think about. The function uses `std::numeric_limits`, so you'll need to include `<limits>`. You didn't *need* to put this in your header file, because it isn't in the function declaration and thus wouldn't be used. Adding unused headers isn't a disaster, but it can slow your build down a little, which can make a difference in a very large codebase. Instead, include the `<limits>` header in `input.cpp`.

You are also using `std::expected`, `std::string`, and `std::istream`, all of which you included in `input.h`, so including the `input.h` file in `input.cpp` will make the includes visible to the source file. Put your own header includes in double quotes, `"`, rather than the angle brackets, `<>`, you use for library headers: `#include "input.h"`.

Second, you've declared `get_number` in a namespace, so you need to put its definition in that namespace too. Pulling this together gives you the code shown in [Example 5-2](#).

Example 5-2. A separate source file called `input.cpp`

```
#include <limits> ❶

#include "input.h" ❷

namespace stock_prices ❸
{
    std::expected<double, std::string> get_number(std::istream & input_stream) ❹
    {
        double number{};
        input_stream >> number;
        if(input_stream)
        {
            return number;
        }
        input_stream.clear();
        input_stream.ignore(
            std::numeric_limits<std::streamsize>::max(),
            '\n'
        );
        return std::unexpected{"That's not a number"};
    }
}
```

- ❶ Includes `<limits>`, a standard library header
- ❷ Includes your header file (notice the quotes rather than angle brackets and your `.h` extension)
- ❸ Opens a namespace
- ❹ Defines the function

Now that you have defined a function in a separate source file, with a declaration in a header, you can use it in any program you write. Just include your header in any source files where you want to use the function, and add the source file to your build instructions. I'll show you how to do that shortly.

First, though, you need to use the function and provide a `main` somewhere. Create a new source file and call it `main.cpp`. Now you're going to write a function based on the `main` function from [Example 4-11](#).

Include your header file, *input.h*, and add a function that will fetch prices. Previously, you wrote this inside the *main* function, getting numbers in a *while* loop, like this:

```
std::vector<double> numbers{};
auto number = get_number(std::cin);
while(number.has_value())
{
    numbers.push_back(number.value());
    std::cout << "Got " << number.value() << " thanks!\n>";
    number = get_number(std::cin);
}
```

Now that you're writing a separate function, you will return a `std::vector<double>` and take a `std::istream` to read numbers from. The `get_number` function is in the namespace `stock_prices`, so you need to specify `stock_prices::` before the function call. Pulling all this together gives you the code in [Example 5-3](#).

Example 5-3. Calling a function in another source file to get several numbers

```
#include <iostream>
#include <vector>

#include "input.h" ❶

std::vector<double> get_prices(std::istream & input_stream)
{
    std::cout << "Please enter some numbers.\n>";
    std::vector<double> numbers{};
    auto number = stock_prices::get_number(input_stream); ❷
    while(number.has_value())
    {
        numbers.push_back(number.value());
        std::cout << '>'; ❸
        number = stock_prices::get_number(input_stream);
    }
    return numbers;
}

int main()
{
    auto prices = get_prices(std::cin); ❹
}
```

- ❶ Includes your header
- ❷ Uses `get_number` from the `stock_prices` namespace
- ❸ Add a `>` to the output so the user knows to enter something else

4 Calls `get_prices` using `std::cin`

To build both source files, specify both their names. Previously, when building code, you’ve used the warning and language version flags and then stated one `.cpp` file, followed by the output. This time, you’ll state two `.cpp` files. For example, if you’re using `g++`:

```
g++ -Wall -std=c++23 input.cpp main.cpp -o stock_prices
```



There are various build systems, such as GNU Make and CMake, that generate build systems. They allow you to use a single instruction without having to remember to list all the relevant source files. They are beyond the scope of this book, but you can find many tutorials on the internet. If you are using an IDE, it will use a build system in the background.

Now you can run your program. Try entering a few numbers, and then finish by entering some nonnumeric input, like “bye”:

```
Please enter some numbers.  
>2  
>3.4  
>-1  
>5  
>4.56789  
>bye
```

Analyzing Your Numbers Using Algorithms

It would be sensible to do something with the numbers you entered, wouldn’t it? You fetched the prices in `main.cpp`, inside the `main` function:

```
auto prices = get_prices(std::cin);
```

Include `<algorithm>` at the top of your `main.cpp` file. This is useful in several ways. C++ *ranges* allow you to use a whole container, or just part of one, easily. Ranges were introduced in C++20, and there are still older versions of algorithms in the library. Some algorithms have both versions, but others aren’t supported by ranges yet.

Let’s look at how to use range algorithms and classic algorithms. You’ll begin by finding the largest and smallest of your values. Previously you found the biggest number in a container using a loop in [Example 4-7](#). Rather than using a range-based for loop here, you can use an algorithm from the C++ standard library called `minmax`, which finds the largest and smallest values. The result has two values, `min` and `max`, telling you the values you want.



If you don't enter any numbers, the range will be empty, and the behavior of `minmax` will thus be undefined. The actual behavior will vary between toolchains.

Add the call to `main` and display the values, provided the range isn't empty:

```
#include <algorithm> ❶
#include <iostream>
#include <vector>

#include "input.h"

// get_prices as before ❷

int main()
{
    auto prices = get_prices(std::cin);
    if(!prices.empty()) ❸
    {
        auto result = std::ranges::minmax(prices); ❹
        std::cout << "min " << result.min << '\n'; ❺
        std::cout << "max " << result.max << '\n'; ❻
    }
}
```

- ❶ Includes algorithms
- ❷ `get_prices` as before
- ❸ Checks that there are prices, not an empty range
- ❹ Finds the biggest and smallest elements
- ❺ Prints the smallest price
- ❻ Prints the largest price

If you build and run your code, you will now see some output:

```
Please enter some numbers.
>2
>3
>5.6
>-9
>bye
min -9
max 5.6
```

You've found the minimum and maximum values.

Using Predicates in Algorithms

The `minmax` algorithm is a *search* algorithm. There are many more, including some that allow you to look for elements that fulfill certain criteria. This is provided by a *predicate*: a function that returns true or false for a value. C++ algorithms provide a few ways to search and filter out values. For example, a negative stock price seems unlikely, so you might want to remove any of those before starting further analysis.

Let's see if any prices have negative values. Create two new files: *analysis.cpp* and *analysis.h*. Then you'll write a predicate to decide if a number is negative. It needs only one line. You could put this predicate function in your new source file, but people often put short functions in headers. When you do this, you are defining them *inline*. You can then copy them directly into calling code without the overhead of a function call, which can make the code run quicker. You also need to add the keyword `inline` before the definition. Otherwise, you could potentially have several copies of the same function in each source file (or translation unit) that includes the header. In this case, you'll add the code to your header and then compare a double against 0.0.



The term *translation unit* refers to a source file and the headers it includes. Each function, variable, or other type must be defined only once in any one translation unit. There must be only one single, unambiguous definition for such things in the whole resulting program. This is called the *one-definition rule* (ODR).

If you don't add `inline` to a function that you place in a header, each C++ file that uses the header will have its own copy of the function, which will break the ODR. Even if you used `#pragma once`, that prevents a header from being used more than once in a single source file. Every source file still includes the headers specified, so it will end up with a copy of the function. Adding `inline` prevents this error. I recommend watching [Roger Orr's talk](#) about the ODR from the ACCU 2024 conference for more details. ACCU (formally the Association of C and C++ Users) is a group of programmers who care about their craft. They still cover C++, but more in addition.

Use the `stock_prices` namespace again in your new *analysis.h* file, like this:

```
#pragma once ❶

namespace stock_prices ❷
{
    inline bool negative(double value) ❸
    {
```

```

        return value < 0.0;
    }
}

```

- ❶ Guards against multiple inclusion
- ❷ Reopens the namespace
- ❸ Defines a short function inline

The predicate `negative` is a *unary* predicate, meaning that it takes just one parameter, `value`.

In your `analysis.cpp` file, include your header file, `analysis.h`, at the top, ready for later additions. Now you can use your `negative` function from `main`.

You'll start by counting any negative prices. Ranges provide a function called `count_if` via the `algorithm` header. This takes a range and a predicate, so you can use your `negative` function. Add the code to `main` like this:

```

#include <algorithm>
#include <iostream>

#include "analysis.h"
#include "input.h" ❶

std::vector<double> get_prices(std::istream & input_stream)
{
    std::cout << "Please enter some numbers.\n>";
    std::vector<double> numbers{};
    auto number = stock_prices::get_number(input_stream);
    while(number.has_value())
    {
        numbers.push_back(number.value());
        std::cout << '>';
        number = stock_prices::get_number(input_stream);
    }
    return numbers;
}

int main()
{
    auto prices = get_prices(std::cin); ❷
    if(!prices.empty())
    {
        auto result = std::ranges::minmax(prices);
        std::cout << "min " << result.min << '\n';
        std::cout << "max " << result.max << '\n';
    }
    auto invalid = std::ranges::count_if(prices, stock_prices::negative); ❸
}

```

```
std::cout << invalid << " prices below zero\n";
}
```

- ❶ Includes your new header
- ❷ Gets prices and finds the min and max as before
- ❸ Calls `count_if` to find negative numbers

Just pause for a moment, and appreciate this single line to count negative prices:

```
auto invalid = std::ranges::count_if(prices, stock_prices::negative);
```

You could write a loop to count these yourself, but doing so requires some thought, and you might make mistakes. Using the algorithm instead gives you clearer code and reduces the possibility of introducing errors. You can also tell what it's doing: counting any negative prices. You could easily count something else using a different predicate.

Build your code again and try it. You still need to use *main.cpp* and *input.cpp* in your build, but the `negative` function is in the *analysis* header, so you don't need to mention the *analysis.cpp* file. Your program will now count any negative numbers you enter, as well as finding the minimum and maximum values:

```
Please enter some numbers.
>1.0
>1.2
>-0.5
>3
>done
min -0.5
max 3
1 prices below zero
```



You made a new source file, *analysis.cpp*, and included the *analysis.h* file. The source file only has the `include` line in it. The function is defined `inline` in the header, so it's available whenever the header file is included. When you add functions into *analysis.cpp*, you need to list the source in the build instructions if you want to use them from `main`. I'll remind you when you need to do this.

Next, you can remove any negative values and then find the average price.

C++20 introduced a function to erase elements according to a predicate, called `erase_if`. It returns the number of elements it erased, so you can use it to count the negative values as you erase them:

```
auto erased = std::erase_if(prices, stock_prices::negative);
std::cout << erased << " prices below zero\n";
```

Some codebases still use older C++ versions, so before moving on, I'll show you how to remove elements from a container without using the `erase_if` function. You will learn more C++ as you read this, even if you decide that `erase_if` is much simpler.

Using Iterators in Algorithms

At the start of “[Analyzing Your Numbers Using Algorithms](#)” on page 89, I mentioned that some algorithms don't use ranges yet. Many do, but it's worth being able to use the older versions as well.

When you called `minmax` and `count_if`, you passed the `prices` container to the algorithms. The older versions of these algorithms take two *iterators*, which you started to learn about in [Example 4-9](#). Many languages have the idea of an iterator: “An *iterator* is an abstract view of a position in a sequence that's independent of both the type of the elements and the sequence itself.”¹ For example, you can create a `std::array` and get `begin` and `end` iterators:

```
std::array whole_numbers{1,2,3};
auto numbers_begin = whole_numbers.begin();
auto numbers_end = whole_numbers.end();
```

You could do likewise with a `std::vector`:

```
std::vector prices{1.01, 2.02, 3.03};
auto prices_begin = prices.begin();
auto prices_end = prices.end();
```

You can use these iterators in any algorithm, even though the containers are different and contain different types of elements.

The iterator indicates the element's position in the container. To refer to the elements, you need to use the asterisk, `*`, called a *dereference* operator, like this:

```
*numbers_begin ❶
*prices_begin   ❷
```

- ❶ Gives the value of the `int` at the position indicated by the `array_iterator`
- ❷ Gives the value of the `double` at the position indicated by the `vector_iterator`

In “[What Happens When You Delete from a Vector](#)” on page 78, you erased a few elements from a vector, starting with the first item. You used `begin` to find an iterator to the first element:

¹ Definition from *C# Brain Teasers: Exercise Your Mind* by Steve Love (Pragmatic Bookshelf, 2025), <https://oreil.ly/fzS1I>.

```
numbers.erase(numbers.begin());
numbers.erase(numbers.begin() + 1, numbers.begin() + 2);
```

By convention, the second iterator is one past the last element on which you want to perform the algorithm. When you didn't provide a second argument, `end` was used. The pair of iterators form a *half-open range*, a term borrowed from mathematics. A mathematical *closed range*, like $[1, 3]$, includes the 1 and 3 at the beginning and end, so that means it includes the whole numbers 1, 2, and 3. In contrast, an *open range*, written with curved brackets, like $(1, 3)$, doesn't include the 1 or 3, just what's between them. So the only whole number it would include is 2. **Figure 5-2** shows square brackets including a number and curved brackets not including the number, for closed, open, and half-open ranges.

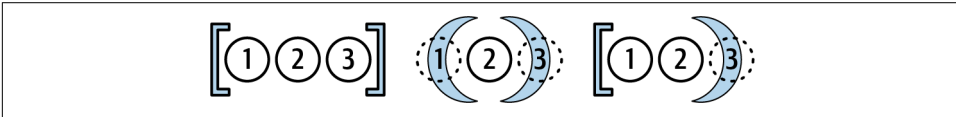


Figure 5-2. Mathematical closed range (1, 2, and 3 included), open range (only 2 included), and half-open range (1 and 2 included)

The pair of iterators used by an algorithm are a half-open range, which includes `begin` and everything up to, but *not* including, `end`. It's written as $[\text{begin}, \text{end})$. `end` returns an iterator positioned past the last item, so the half-open range means everything from `begin` up to the last item. (You don't need to know the math terminology, but it does get used in documentation from time to time.)

You saw the `begin` and `end` of a `std::vector` in **Figure 4-4**. Any algorithm that takes two iterators starts at the first and stops when it reaches the second, without using that second iterator. However, the iterators don't need to be `begin` and `end`: instead of the whole container, you could use part of the container.

Recall how you used `count_if` from `ranges`:

```
auto invalid = std::ranges::count_if(prices, stock_prices::negative);
```

You can pass your prices' `begin` and `end` elements to the older function instead:

```
auto invalid =
    std::count_if(prices.begin(), prices.end(), stock_prices::negative);
```

Both approaches operate on the whole container.

The Old Way to Remove Items

The standard library algorithms also provide `remove_if` functions, both for ranges and for pairs of iterators. You might see them used, so it's worth knowing what they do and why.

You'll use the iterator version in this section, and you can try the range version yourself afterward. Let's start with a failing test. Add a function in your *analysis.cpp* file, called `remove_invalid`, along with a test function. The test can use a `std::vector` with one invalid and one valid value, like `{-1.2, 3.5}`:

```
#include <algorithm>
#include <cassert> ❶

#include "analysis.h" ❷

namespace stock_prices ❸
{
    std::vector<double> remove_invalid(std::vector<double> prices) ❹
    {
        return prices;
    }

    void test_analysis() ❺
    {
        auto got = remove_invalid({-1.2, 3.5}); ❻
        assert(got.size() == 1);
        assert(got[0] == 3.5);
    }
}
```

- ❶ Includes C's `assert` function
- ❷ Includes your own header
- ❸ Reopens the `stock_prices` namespace
- ❹ Defines a function to remove invalid elements (which does nothing, yet)
- ❺ Defines the tests
- ❻ Uses an initializer list with two values to make a `std::vector`

You're using the `cassert` header, as you did before in [“Starting with a Failing Test” on page 24](#), to validate your code. The `remove_invalid` just returns a copy of the original prices at the moment, so the test will fail.

Add a declaration in *analysis.h* for the new function and for your tests, in the namespace:

```
namespace stock_prices
{
    inline bool negative(double value)
    {
        return value < 0.0;
    }
}
```

```

    }
    std::vector<double> remove_invalid(std::vector<double> prices); ❶
    void test_analysis(); ❷
}

```

❶ Declares the removal function

❷ Declares a test function

Now call `test_analysis` from `main`. It's in a namespace, so you need to specify that, too. Put the call inside `main`:

```

int main()
{
    stock_prices::test_analysis(); ❶
    //... ❷
}

```

❶ Calls the tests

❷ As before

Build your code again, adding *analysis.cpp* to the source files this time. For example, with `g++`:

```
g++ -Wall -std=c++23 analysis.cpp input.cpp main.cpp -o stock_prices
```

When you run your code, the test will fail, because `remove_if` does not remove the elements. You therefore see an assertion failure:

```
Assertion 'got.size() == 1' failed.
```

Let's make the test pass. You will use `std::remove_if`. The call itself is similar to `count_if`:

```

auto something =
    std::remove_if(prices.begin(), prices.end(), stock_prices::negative);

```

What will the algorithm return? To answer that question, let's consider what this algorithm actually does.

Despite its name, it doesn't actually remove any elements. Your container will remain the same size! Instead, this algorithm shifts the elements you want to keep toward the beginning of the container. The elements you want to keep start at `begin` and go up to a new `end`, which is the returned value. So, the mysterious `something` in the previous code is an iterator, pointing one past the end of the elements you want. What is left after that is *unspecified*, meaning each toolchain can decide what happens. You can then use the returned value instead of the original `end` to refer to the elements you want. [Figure 5-3](#) shows what happens.

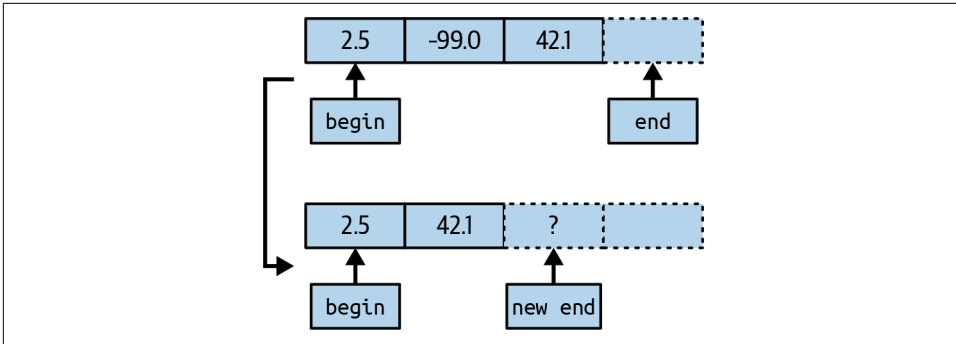


Figure 5-3. Removing negative elements

This isn't exactly intuitive at first sight. However, the original algorithms take pairs of iterators, because doing so is more general than providing an implementation of each algorithm for each container. Though the algorithm can move elements around, it cannot adapt the size of the container. Since `std::remove_if` returns the new end, your calling code can use a subsequent call to `erase` to change the container directly.

Let's make the test pass. The removal function needs to return a vector with one fewer element for the test case. See if you can get most of the way to writing this function on your own. If not, use the code in [Example 5-4](#).

Example 5-4. Removing invalid elements properly

```
#include <algorithm>
#include <cassert>

#include "analysis.h"

namespace stock_prices
{
    std::vector<double> remove_invalid(std::vector<double> prices)
    {
        auto new_end = std::remove_if(prices.begin(), prices.end(), negative); ❶
        prices.erase(new_end, prices.end()); ❷
        return prices;
    }
    void test_analysis()
    {
        auto got = remove_invalid({-1.2, 3.5});
        assert(got.size() == 1);
        assert(got[0] == 3.5);
    }
}
```

- ❶ Includes C's `assert` function
- ❷ Includes your own header

Did you notice that the `remove_invalid` function uses pass by value to take the prices as a copy? Perhaps using pass by value seems more sensible now. You have passed by reference before using a `const &`, for example, in [Example 3-2](#). The `const` means you won't change the values, and the reference, `&`, avoids copying the data. This time, though, you *do* want to change the values. The calling code might want to keep the original values, so returning a new vector is better than mutating the original. You could pass a `const &` and then copy the vector yourself, but passing the parameter by value gives you a copy automatically.

Your container will be the same size after you call `remove_if`, so you need to also call `erase` starting at `new_end`. CppReference states that a call to `remove` is typically followed by a call to `erase`, which is called the *erase-remove idiom*. This call erases unwanted (and unspecified) values. If you don't erase them, you would need to keep track of the returned iterator to avoid using them.

If you build and run your code now, your test will pass.

You have seen the newer and older ways to remove elements from a container, and you'll practice more algorithms in [Chapter 6](#). Now that you only have valid prices, however, you're ready to find the average.

Finding an Average with an Algorithm

You'll add the following code to `analysis.cpp`, and you will be able to reuse it in later chapters. The *arithmetic mean* of a collection of numbers is their total divided by their count. You can use a for loop to achieve this:

```
double average(const std::vector<double> & prices)
{
    double sum{0.0};
    for(const double & price: prices)
    {
        sum += price;
    }
    return sum/prices.size();
}
```

There's nothing wrong with a `for` loop, but sometimes the algorithm version will be more efficient or deal with edge cases better. Let's use an algorithm instead. Before we do, can you think of an edge case? It's always worth starting to think of potential issues when you write code. Sketching them out in scenarios (or even unit tests) will remind you to deal with them. For instance, what happens if the `prices` container is empty? Dividing by zero is never a good idea. You could indicate a problem by throwing an exception or using a `std::expected`, or you could return 0.0.

Let's use an exception for practice. Add a declaration for your upcoming function to *analysis.h*, inside the namespace:

```
double average(const std::vector<double> & prices);
```

You'll add the definition inside *analysis.cpp*, as well as a couple of tests. An empty vector and a vector with one element will cover what you need to get started. You can do this in stages, starting once again with a failing test. Start with a basic (but wrong) average function in *analysis.cpp*, inside the namespace:

```
double average(const std::vector<double> & prices)
{
    return 0.0;
}
```

Add both tests to your `test_analysis` function. The first test will *try* to find the average of an empty vector. The second will assert that the average of a number is the number itself. [Example 5-5](#) shows what you need.

Example 5-5. Two new tests for average

```
void test_analysis()
{
    auto got = remove_invalid({-1.2, 3.5});
    assert(got.size() == 1);
    assert(got[0] == 3.5);

    try
    {
        average({}); ❶
        assert(false); ❷
    }
    catch(const std::exception &) ❸
    {
    }

    assert(average({1.0})==1.0); ❹
}
```

- ❶ Tries to find the average of an empty vector, {}.

- ❷ This line should never be reached, since you expect an exception, so the program asserts if you get here.
- ❸ Catches any `std::exception`, so if this happens, the test has succeeded.
- ❹ Asserts that the average of `{1.0}` is `1.0`.

If you run this now, you will see a message like `Assertion false failed`. Because your function doesn't check the prices, no exception will be thrown for an empty vector. So at the top of `average`, add a check for an empty vector, and have it throw a `std::invalid_argument` error with a suitable message if needed. The `std::invalid_argument` exception lives in the `<stdexcept>` header, so include that too, near the top of your *analysis.cpp* file:

```
#include <stdexcept> ❶
//.. ❷
namespace stock_prices
{
    //... ❸
    double average(const std::vector<double> & prices)
    {
        if(prices.empty())
            throw std::invalid_argument("Prices cannot be empty"); ❹
        return 0.0; ❺
    }
}
```

- ❶ Includes various exception types, such as `std::invalid_argument`
- ❷ As before
- ❸ Also as before
- ❹ Throws an exception if there are no values
- ❺ Returns `0.0` (which isn't quite there yet)

Now you will see a new message like:

```
Assertion 'average({1.0})==1.0' failed.
```

You have dealt with the edge case of an empty vector. Now you need to implement code for actual values. You can use the `accumulate` function from the standard library for this. This algorithm lives in the `<numeric>` header. There are two overloads. You'll use the first, which takes `begin` and `end` iterators and an initial value, and sums the elements and the initial value.

Again, the average is the sum of all the numbers, divided by the count. You'll use `prices`' `begin` and `end` and provide an initial value of `0.0`, using `double{}`. Then `prices.size()` tells you how many you have. Instead of returning `0.0`, you now return:

```
return std::accumulate(prices.begin(), prices.end(), double{})/prices.size();
```



You saw how to use a for loop to sum numbers at the start of this section. You can use `std::accumulate` instead. There's not much difference in this case, but practicing algorithms is useful. They are often simpler to use and help you avoid common mistakes, so using them is therefore considered a best practice.

If you build your code again and run it, your tests will pass.

You have added a little code at a time, testing at regular points. The *analysis.cpp* file should now include several headers, two functions, and some test code, as shown in **Example 5-6**.

Example 5-6. Full analysis source

```
#include <algorithm>
#include <cassert>
#include <numeric>
#include <stdexcept>

#include "analysis.h"

namespace stock_prices
{
    std::vector<double> remove_invalid(std::vector<double> prices)
    {
        auto new_end = std::remove_if(prices.begin(), prices.end(), negative);
        prices.erase(new_end, prices.end());
        return prices;
    }

    double average(const std::vector<double> & prices)
    {
        if(prices.empty())
            throw std::invalid_argument("Prices cannot be empty");
        return std::accumulate(prices.begin(),
                                prices.end(),
                                double{})
            /prices.size();
    }

    void test_analysis()
    {
```

```

    auto got = remove_invalid({-1.2, 3.5});
    assert(got.size() == 1);
    assert(got[0] == 3.5);

    try
    {
        average({});
        assert(false);
    }
    catch(const std::exception &)
    {
    }

    assert(average({1.0})==1.0);
}

```

You can call `average` from `main` to display the average. Don't forget the namespace:

```
std::cout << "Average " << stock_prices::average(prices) << '\n';
```

Though the average doesn't tell you much, you can track how it changes over time to (attempt to) predict if values will go up or down and therefore to decide whether to buy or sell stock.

Understanding Algorithms in More Depth

You have used some range algorithms and some classic algorithms with `begin` and `end`. You have also seen how you might use a loop instead, but using a raw loop can cause problems. If you use a loop while you erase elements, what happens?

Using for Loops

You've used a range-based `for` loop a few times now. However, there's another type of `for` loop, sometimes called a *C-style* `for` loop. This `for` loop has three parts:

- A starting statement, such as an iterator, at `begin`
- A condition to tell the loop when to stop, such as the iterator matching `end`
- An *iteration expression*, which is executed after the loop body: for example, incrementing the iterator so it's ready for the next time around the loop

To loop over the prices, you use a `for` loop like this:

```

for(auto iterator = prices.begin(); ❶
    iterator != prices.end(); ❷
    ++iterator) ❸
{
}

```


- ❶ Starting statement
- ❷ Ending condition, stopping the loop when this is false
- ❸ Increments the iterator

Like the `while` loop you used in [Example 4-2](#), and like range-based `for` loops, you put statements in the loop body, `{}`.

Try removing the negative numbers in your loop. If you want to try the code, put it in *analysis.cpp*. Notice that the `negative` function takes a `double`, rather than an iterator to a `double`. Armed with the iterator, you can use the dereference operator `*`, which you met in “Using Iterators in Algorithms” on page 94, to get the value itself. So `*iterator` tells you the `double` in the vector at `iterator`.

If your loop tries to erase an element in a vector, what do you think will happen? Find out in [Example 5-7](#).

Example 5-7. A very bad idea

```
std::vector<double> remove_invalid_badly(std::vector<double> prices)
{
    for(auto iterator = prices.begin(); iterator != prices.end(); ++iterator)
    {
        if(negative(*iterator)) ❶
            prices.erase(iterator); ❷
    }
    return prices;
}
```

- ❶ Dereferences iterator to get the double
- ❷ Erases the element

If you try this code, you will see an error. For example, `g++` on [Godbolt reports](#) :

```
Program terminated with signal: SIGSEGV
```

CppReference says that references to the elements at or after the point of the erasure are *invalidated*. When you change a container’s size, the position indicated by the iterator might no longer be there. Trying to increment an invalid iterator is a very bad idea.

The `erase` function actually returns an iterator, telling you the iterator following the last removed element. You can store it with the following code:

```
iterator = prices.erase(iterator);
```

However, this doesn't solve all your problems.

If the end of your range is a negative number, say {1.5, 3.2, -1.5}, the `iterator` will then be at the end. That means the loop increment will go *one past* the end! This is invalid and causes undefined behavior. You can fix this by saving the iterator after an erase and incrementing it only for nonnegative numbers, as follows:

```
std::vector<double> remove_invalid(std::vector<double> prices)
{
    for(auto iterator = prices.begin();
        iterator != prices.end();
        ❶
    )
    {
        if(negative(*iterator))
            iterator = prices.erase(iterator); ❷
        else
            ++iterator; ❸
    }
    return prices;
}
```

- ❶ Leaves the for loop's iterator expression empty
- ❷ Saves the new iterator after an erase
- ❸ Increments `iterator` for nonnegative values

The code now works. Try it yourself or use this [Godbolt](#). It's true that `std::erase_if` was much less to pay attention to, but you have learned a few new things.

If you do use a raw for loop, you need to be very careful. To sum up the issues laid out here, raw loops:

- Are (much) more error-prone
- Often require you to write (much) more code
- Take (much) more time to write, read, and understand

Furthermore, raw loops require more testing than a library algorithm, which has already been tested. Sometimes a raw loop can take more time to run, too, and can therefore be more expensive.

Using a raw loop for an algorithm that already exists can be informative. However, if you catch yourself writing your own sorting algorithm, avoid reinventing the wheel by checking if C++ already has it. Its algorithms have been designed to work with any container, and almost anything you want is likely to be there for you already.

Binary Operators and Predicates

So far, you've used your negative unary predicate a few times. Some algorithms take *binary* predicates, which are functions that take two parameters and return a `bool`, like this:

```
bool some_function(double x, double y);
```

Let's see an example. You can sort your prices, using `std::sort`, from the algorithm header. This returns `void`, because it mutates the collection. Add your code in `main`:

```
std::sort(prices.begin(), prices.end());
for(auto p: prices)
{
    std::cout << p << '\n';
}
```

By default, the elements are sorted in increasing order:

```
Please enter some numbers.
>4
>3
>10.2
>done
3
4
10.2
```

You can change the ordering by providing a binary predicate to compare elements. The comparison takes two parameters and returns a `bool`. You can change the comparison you use, for example, to put your prices in descending order.

You could write a named function to decide if one number is greater than another, but you could also do that using a function object from the `<functional>` header. A *function object* is a class with an operator `()`, often referred to as the *call operator*. The `<functional>` header contains several function objects, including `std::greater`. This is a template, so you can use it for any type.

You'll now pass a `std::greater` function object to `std::sort`. Don't forget to include the `<functional>` header:

```
std::sort(prices.begin(), prices.end(), std::greater{});
```

Your prices are now sorted in descending order:

```
Please enter some numbers.
>4
>3
>10.2
>done
10.2
```

4
3

If you want to use ranges instead, the calls are similar, but use the container rather than two iterators. You'll also need to use the ranges' greater function object:

```
std::ranges::sort(prices); ❶  
std::ranges::sort(prices, std::ranges::greater{}); ❷
```

❶ Sorts in increasing order

❷ Sorts in decreasing order

Because C++ is always evolving, you'll frequently find that there's more than one way to achieve what you want. Making use of its higher-level concepts, like ranges and algorithms, often leads to shorter code and can help you avoid problems; still, it's worth being aware of some of the alternatives, in case you come across them in older codebases.

More on Iterators

Once the prices are sorted in decreasing order, you can find the first negative value, using `std::ranges::find_if`:

```
auto iterator = std::ranges::find_if(prices,  
                                     stock_prices::negative);
```

You can use this iterator to make a new `std::vector`:

```
auto positive = std::vector(prices.begin(), iterator);
```

This constructs a new `std::vector` by copying the elements from `begin` up to, but not including, the first negative number. As you saw in “[Initializing a Vector with a Fixed Value](#)” on page 79, you can use curly braces to give specific values, but parentheses do something completely different. In this case, you are providing two iterators to specify a range of elements.

You could also sort the elements in ascending order and copy from the iterator returned from `std::find_if` to the end of the collection. Lots of options! You've seen a few ways to remove negative prices now. Sorting is useful, but putting the stock prices in order means you lose information about their behavior over time. To correct for that, you will try some more analysis in the next chapter.

Conclusion

This chapter introduced some standard algorithms, but you learned other parts of C++ too. You built a larger program using more than one source file and wrote your own header files. You will be able to reuse the input and analysis source files later in

this book. You used a namespace to structure your code. You also defined a function `inline` in a header and learned about C-style for loops.

You started with ranges, finding the largest and smallest values using `minmax`. You tried the older iterator-based algorithms, too, and you used `begin` and `end` to use all the items in a container. Key takeaways include:

- An iterator indicates a position in a sequence.
- The end is really a one-past-the-end iterator.
- You use operator `*` to dereference an iterator, getting the value at its position.
- Incrementing an iterator moves to the next element in a container, but you must not go beyond the end.

You learned how to write unary and binary predicates and use them in algorithms, too. Among other things, you learned that:

- Predicates return a `bool`.
- A unary predicate takes one parameter, for example, deciding if a number is negative.
- A binary predicate takes two parameters, for example, deciding if one number is greater than another.

You used a named function, `negative`, and the function objects `std::greater` and `std::ranges::greater` from the `<functional>` header. **Chapter 6** will show you other ways to write predicates and other functions you can use in algorithms.

C++ frequently provides more than one approach. Using algorithms can lead to neater code than C-style for loops and can be less verbose than using two iterators. Whichever approach you use, always test your code and think about possible edge cases (like if the container is empty).

Lambdas and the Ranges Library

You used a few algorithms in [Chapter 5](#). Some take a function, and you used a hand-written negative function to find negative numbers. Using a named function is fine, but C++ provides alternatives. For example, you also used function objects, including `std::greater`. C++ gives you another way to provide functions to algorithms: using *lambdas*, or *anonymous* (unnamed) functions. Lambdas offer a concise way to write short functions, letting you write the predicates and other functions directly in the call to the standard algorithms.

This chapter will show you how to write lambdas and give you further practice using algorithms. You used some range algorithms in the previous chapter. Ranges also offer several library features, including views, so you'll see how to use some of these. You'll also learn in this chapter how to use lambdas for more than the standard library algorithms.

In [Chapter 5](#), you wrote a function called `get_prices` in `main`, in [Example 5-3](#). This used the `get_number` function you defined in *input.cpp*. In this chapter, you will write a more general-purpose `get_prices` function, which you can use in subsequent chapters, as well as a couple of trading strategies. They won't make you much money but will show you more C++. By the end of this chapter, your analysis and input source files will be more useful, and from there, [Chapter 7](#) will show you how to generate prices rather than typing them in.

Removing Negative Numbers Using a Lambda

In [Chapter 5](#), you used `erase_if` to remove negative numbers:

```
auto erased = std::erase_if(prices, stock_prices::negative);  
std::cout << erased << " prices below zero\n";
```

You put the negative function in your *analysis.h* file:

```
inline bool negative(double value)
{
    return value < 0.0;
}
```

An alternative is to use an anonymous, or *lambda*, function,¹ directly in the `erase_if` call. Unlike the functions you have written so far, a lambda doesn't have a name, hence the alternative name "anonymous," and its return type can be deduced from its definition. This leaves the parameters and implementation:

```
(double value){ return value < 0.0; }
```

That's almost a lambda. You need one more part.

A lambda can use variables in the surrounding scope, either by reference or by value. If you need to use any, they go in square brackets `[]` at the start. You don't need any other variables to detect whether a value is negative, so leave the brackets empty:

```
[](double value){ return value < 0.0; }
```

You can pass this lambda directly into `erase_if`:

```
auto erased = std::erase_if(prices, [](double value){ return value < 0.0; });
```

Using the named function `negative` makes it clear what the code will do, but putting the implementation directly in the algorithm call can have advantages, too. For example, you can see exactly what the code is doing without having to look somewhere else for a function implementation. Lambdas are a concise way to define short functions. For functions that need more than a couple of lines of code, prefer a named function.

Let's recap:

- Lambdas are functions but have no names.
- The return can be deduced for you.
- Lambdas start with `[]`, which indicates variables from the surrounding scope that the lambda needs.
- Lambdas take parameters, like named functions do.
- Lambdas have an implementation in `{ }`, like named functions do.
- A lambda is a clear, concise way to pass a function to an algorithm.

You can also assign a lambda to a variable and use that in `erase_if`:

¹ Lambdas are technically known as *callable*s, a more general idea than a function.

```
auto lambda = [](double value){ return value < 0.0; };
auto erased = std::erase_if(prices, lambda);
```

Each lambda has its own unique type

If you assign a lambda to a variable, use `auto`. Every lambda has its own unique type, which C++ creates. You can declare two lambdas with the same implementation:

```
auto first_lambda = [](double value){
    return value < 0.0;
};
auto second_lambda = [](double value){
    return value < 0.0;
};
```

The types of `first_lambda` and `second_lambda` are different.

You can call the lambda yourself, passing a parameter in parentheses as you would for the `negate` function:

```
bool invalid = lambda(-67.0);
```

Create a new `main.cpp` file and try using a lambda:

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector prices{1.01, 2.02, 3.03, -4.04};
    auto lambda = [](double value){ return value < 0.0; };
    auto erased = std::erase_if(prices, lambda);
    std::cout << erased << " prices below zero\n";
}
```

Let's use lambdas a bit more.

Using a Lambda to Vary Behavior via `std::function`

In this section, you're going to build on the input and analysis code you started in [Chapter 5](#). You can add to that code or start new files. If you want a fresh version, use [Example 5-1](#) and [Example 5-2](#). Let's start with input.

You can generalize the `get_prices` function you wrote in [Example 5-3](#). To refresh your memory, [Example 6-1](#) shows that code.

Example 6-1. A reminder of the `get_prices` function

```
std::vector<double> get_prices(std::istream & input_stream)
{
    std::cout << "Please enter some numbers.\n>";
    std::vector<double> numbers{};
    auto number = stock_prices::get_number(input_stream);
    while(number.has_value())
    {
        numbers.push_back(number.value());
        std::cout << '>'; ❶
        number = stock_prices::get_number(input_stream);
    }
}
```

❶ Adds a > character to prompt for more input

You used a general input stream, `istream`, here so that you could add some tests using a string stream rather than `std::cin`. However, we didn't add tests in the previous chapter. The function currently outputs messages to the screen, so a test would just spew messages onto the screen, which isn't helpful. Ideally, you only want to see *if* tests pass or fail and *why*. Extra output is noisy and distracting.

You can send in a function instead of calling `std::cout` directly, using a named function or a lambda. This function will replace the `std::cout` line, allowing you to decide whether you want to print output or do nothing. The function doesn't need parameters, and it can return `void`. A suitable signature would be:

```
void prompt();
```

How do you send a function like this to `get_prices`? In “Each lambda has its own unique type” on page 111, you saw that each lambda has a unique type. This means you can't specify a generic type for a function parameter, suitable for any lambda. You could use a template, but we'll get to that in Chapter 15. For now, you can use a class template called `std::function` from the `<functional>` header, which is a general-purpose way to store any function.



A `std::function` makes a copy of the given function. This is less efficient than using the function directly, but it's a reasonable choice for code you will call only once or twice. If you need something to happen thousands of times per second, though, you should investigate alternatives, like templates.

Use the familiar `<>` for a template. Add the return type, `void`, and `()` for parameters, like this:

```
std::function<void ()> prompt;
```

The `void ()` looks like the prompt signature but has no name between the return and the parameters. The new version of `get_prices` looks like this:

```
std::vector<double> get_prices(std::istream & input_stream,  
                               std::function<void ()> prompt);
```

You can use a named function or a lambda for the prompt. From `main`, you will use a lambda to prompt with a `>` and use `std::cin`:

```
auto prompt = [] () { std::cout << '>'; };  
auto prices = stock_prices::get_prices(std::cin, prompt);
```

You will do something else from the tests, using an empty lambda to avoid writing to `std::cout`. Using `std::function` has given you options.

Add the more general `get_prices` function in the input files. This process has four steps:

1. Add the declarations of `get_prices` and `test_input` to *input.h*.
2. Add the definition of `get_input` to *input.cpp*.
3. Add the tests to *input.cpp*.
4. Call the new function from `main`.

Add the declaration of `get_prices` to your *input.h* file. Declare a `test_input` function, too, as shown in [Example 6-2](#).

Example 6-2. Additions to input.h

```
#pragma once  
  
#include <expected>  
#include <functional> ❶  
#include <istream>  
#include <string>  
#include <vector> ❷  
  
namespace stock_prices  
{  
    std::expected<double, std::string> get_number(std::istream & input_stream);  
    std::vector<double> get_prices(std::istream & input_stream,  
                                   std::function<void ()> prompt); ❸  
  
    void test_input(); ❹  
}
```

- ❶ Includes `functional` for `std::function`

- ❷ Includes `std::vector` used in the `get_prices` function declaration
- ❸ Declares `get_prices` function
- ❹ Declares a test function

The new function goes in *input.cpp*, along with the tests. Replace the `std::cout` lines from [Example 6-1](#) with a call to `prompt()`, as shown in [Example 6-3](#).

Example 6-3. Additions to input.cpp

```
#include <limits>

#include "input.h"

namespace stock_prices
{
    std::expected<double, std::string> get_number(std::istream & input_stream)
    {
        double number{};
        input_stream >> number;
        if(input_stream)
        {
            return number;
        }
        input_stream.clear();
        input_stream.ignore(
            std::numeric_limits<std::streamsize>::max(),
            '\n'
        );
        return std::unexpected{"That's not a number"};
    }

    std::vector<double> get_prices(std::istream & input_stream,
        std::function<void ()> prompt) ❶
    {
        prompt(); ❷
        std::vector<double> numbers{};
        auto number = stock_prices::get_number(input_stream);
        while(number.has_value())
        {
            numbers.push_back(number.value());
            prompt(); ❸
            number = stock_prices::get_number(input_stream);
        }
        return numbers;
    }
}
```

- ❶ Defines your new `get_prices` function, taking a way to prompt
- ❷ Calls the prompt
- ❸ Calls the prompt again

Now you can add a test function. As before, you need to include `<cassert>`. What would a suitable prompt be? You could use a lambda to write to `std::cout`:

```
auto prompt = [] () { std::cout << '>'; };
```

However, you don't need to see that in your tests. A function that does nothing would be better. Removing the statement between the braces leaves a lambda with no operations to perform. Let's call this a *no op* for short:

```
auto prompt = [] () {};
```



`[](){}` was originally the shortest lambda you could write. You no longer need the parentheses for parameters, `()`, if you have none, so the shortest possible lambda is now `[]{}.`

As you have done before, you can use a string stream to test input. Add a small test function after your `get_prices` function in *input.cpp*, like this:

```
#include <cassert> ❶
#include <limits>
#include <sstream> ❷

#include "input.h"

namespace stock_prices
{
    // As before ❸
    void test_input() ❹
    {
        std::stringstream no_input{""}; ❺
        auto no_op = [](){}; ❻
        assert(get_prices(no_input, no_op).empty()); ❼

        std::stringstream some_input{"1"}; ❽
        assert(get_prices(some_input, no_op).size() == 1); ❾
    }
}
```

- ❶ Includes C's `assert` function
- ❷ Includes string stream
- ❸ Functions `get_number` and `get_prices` as before
- ❹ Defines some tests in a function
- ❺ Makes an empty input stream
- ❻ Defines a lambda that does nothing, to use instead of a prompt
- ❼ Checks for an empty vector
- ❽ Makes an input stream with a single digit
- ❾ Checks for a single digit

The last step is calling your new function from `main`, so you can analyze some input. If you remove your previous code from `main`, you can add new code to test your input function and then call it. Now you'll need to add the message "Please enter some numbers" before the call, since the prompt doesn't include it:

```
#include <iostream>

#include "input.h"

int main()
{
    stock_prices::test_input();

    std::cout << "Please enter some numbers.\n";
    auto prompt = [] () { std::cout << '>'; };
    auto prices = stock_prices::get_prices(std::cin, prompt);
    std::cout << "Got " << prices.size() << " price(s) \n";
}
```

Build your code and try it. Don't forget to add *input.cpp* and *main.cpp* to your instructions. If they are in different directories, include their paths.

When you run your code, the tests will pass, and you will be prompted for numbers:

```
Please enter some numbers.
>
```

As you have done before, you can enter a few numbers and type something non-numeric when you are done. Though the program seems similar when you use it, you have improved your code. You have reused some code and made a more general

function to get prices using `std::function`. This isn't only a better version—it's also easier to test. Let's extend the analysis code next, learning more about ranges.

Filtering Out Negative Numbers Using the Ranges' View

In this section, you will add to your *analysis.cpp* and *analysis.h* files. If you want a fresh copy, look back at [Example 5-6](#) for the source file. The header file contains an inline definition and three declarations:

```
#pragma once

#include <vector>

namespace stock_prices
{
    inline bool negative(double value)
    {
        return value < 0.0;
    }
    std::vector<double> remove_invalid(std::vector<double> prices);
    double average(const std::vector<double> & prices);
    void test_analysis();
}
```

You've learned how to remove negative numbers and return a new copy of your elements. You can tell that from the signature:

```
std::vector<double> remove_invalid(std::vector<double> prices);
```

The function takes `prices` by value, so the `prices` are copied, and returns another vector without negative numbers.

You also used `std::erase_if`, which mutates the original container. However, changing or copying the original data might not be ideal—you might need it later. Furthermore, when you want only a summary statistic, like the mean, copying elements takes time and memory. While this isn't a problem when you're dealing with a small amount of data, you have an alternative.

You've used a few range algorithms. There's a ranges library too, which extends the algorithms. The ranges library provides a *view*, which you can use to avoid copying data or changing a container's contents. The library does this by providing an abstraction over iterators. You can also chain views together. Like rose-tinted glasses, a view adapts what you *see*, not what you are looking at, as indicated in [Figure 6-1](#).

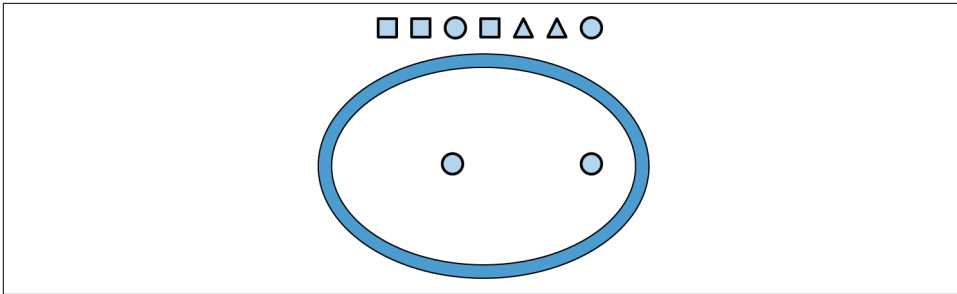


Figure 6-1. A view that filters some elements so that you only see circles

You are going to filter out negative numbers, so you'll use the view's `filter` function. You therefore need to include the `<ranges>` header. The `filter` function takes two parameters: the first tells it what to filter, and the second shows how to filter the elements. You can use a lambda to select non-negative numbers, like this:

```
auto valid_prices = std::views::filter(prices,
    [](double p) { return p >= 0.0; } );
```

Add the filtered view to `main` and display what you get:

```
#include <iostream>
#include <ranges>
#include <vector>

#include "input.h"

int main()
{
    stock_prices::test_input();

    std::cout << "Please enter some numbers.\n";
    auto prompt = [] () { std::cout << '>'; };
    auto prices = stock_prices::get_prices(std::cin, prompt);
    std::cout << "Got " << prices.size() << " price(s) \n";

    std::cout << "The following are valid:\n"; ❶
    auto valid_prices = std::views::filter(prices, ❷
        [](double p) { return p >= 0.0; } ); ❸
    for(double price : valid_prices) ❹
    {
        std::cout << price << '\n';
    }
}
```

- ❶ Displays a message to explain the output
- ❷ Filters prices, giving another view

❸ The lambda finds valid numbers

❹ Loops over valid prices, displaying them

Build and run your code and then enter a few numbers, stopping with something non-numeric. Your program will tell you how many entries you gave and shows the ones that are valid prices:

```
Please enter some numbers.  
>3.55  
>-1  
>9.45  
>1  
>stop  
Got 4 price(s)  
The following are valid:  
3.55  
9.45  
1
```

You've done that before, but this time, you have some tests for the input.

Let's do something with those filtered prices. Include "analysis.h" in main, so you can use your average function. Your function uses a `std::vector<double>`, which you can create explicitly from the view. C++23 introduced `std::ranges::to`, to create a container from a view. This is a function template, so put the `std::vector` you want inside the `<>` and call the function using `()`:

```
std::ranges::to<std::vector>(valid_prices)
```

The compiler deduces that the vector is a `std::vector<double>` because the `valid_prices` are doubles. Try this in your main function, after you display the valid prices:

```
const auto valid_prices_as_vector = std::ranges::to<std::vector>(
    valid_prices
); ❶
const double mean = stock_prices::average( ❷
    valid_prices_as_vector
);
std::cout << "with average " << mean << '\n';
```

❶ Creates a vector from the view

❷ Calls average from the stock_prices namespace



`std::ranges::to` was added in C++23. If your toolchain doesn't support it yet, you can create the vector yourself using `begin` and `end`, like this:

```
std::vector<double> valid_prices_as_vector{
    valid_prices.begin(),
    valid_prices.end()
};
```

Don't forget to include the analysis header in *main.cpp* and to include *analysis.cpp* in your build. Try a few numbers and see what happens. Here's an example run:

```
Please enter some numbers.
>1.25
>3.25
>q
Got 2 price(s)
The following are valid:
1.25
3.25
with average 2.25
```

At the moment, your `average` function takes a vector. Constructing a vector from the view means you've copied elements. You have learned more about ranges, though, so let's try some further analysis.

Using Lambda Captures for Fun and Profit

Suppose you buy stock at the first price you see. You can sell as soon as the price goes above that and make a profit. This is a simplification, of course: in real life, you would be charged for this transaction. Furthermore, the stock's price might never go above the price you paid, so you might never make a profit. However, you're here to learn C++, not to make a profit, and you won't lose any money in a simulation.

Let's add a function to *analysis.cpp* and a declaration to *analysis.h*. The new function will take a `std::vector<double>` of prices by `const` reference and return a potential profit. In finance jargon, when the price goes up, it is described as an *uptick*. You want the first uptick, so add a function declaration to *analysis.h*, inside the `stock_prices` namespace:

```
double profit_on_first_uptick(const std::vector<double> & prices);
```

Given some valid prices and at least one element, you can find the first element using `front`:

```
const double first = prices.front();
```

You must have some elements to do this, so check that the prices are not empty first. You then have the first price. Time to see if you can make a profit.

Use `std::ranges::find_if` to find if there's a price greater than the first. (You've used this algorithm before, in [“More on Iterators” on page 107](#).) The `find_if` function needs a predicate. You want to detect if any price is greater than the first price, because the potential profit comes when the price first goes up. The predicate is called for each value in the container or range, so the predicate will take a double. You can use a lambda to compare the double with the first value, like this:

```
[](double price)
{
    return price > first;
}
```



You can write the lambda on a single line, as you did before:

```
[](double price) { return price > first; }
```

Both versions are equivalent. Notice how a lambda looks exactly like a function, but without a name or an explicit return value.

Unfortunately, this lambda will not compile yet. So far, you can check that you have prices and find the first element with `front`.

Let's look at the lambda:

```
if(prices.empty())
    throw std::invalid_argument("Prices cannot be empty");
const double first = prices.front();
auto lambda = [](double price)
{
    return price > first; ❶
}
```

❶ Tries to use the first price

The lambda uses `first`, but that's declared outside the lambda, so it isn't in scope. To fix the problem, you need to add something *inside* the `[]`. I mentioned earlier that a lambda can use variables in the surrounding scope, either by reference or by value. You can put a specific variable in the braces. The braces *capture* the variable by value, meaning that the lambda can use a copy of the variable:

```
[first](double price) { return price > first; }
```

Let's implement the new `profit_on_first_uptick` in `analysis.cpp`. You can put it anywhere inside the `stock_prices` namespace.

Next, you want to find a value greater than the first and report the difference. If there is none, that means no profit. Remember that the `find_if` algorithm returns an iterator, so you need to dereference the iterator (get its value) using operator `*` to

obtain the first profitable value. Pulling this together gives you a new analysis function, as shown in [Example 6-4](#).

Example 6-4. Function to find potential profit

```
#include <stdexcept>

namespace stock_prices
{
    double profit_on_first_uptick(const std::vector<double> & prices)
    {
        if(prices.empty())
            throw std::invalid_argument("Prices cannot be empty"); ❶
        const double first = prices.front(); ❷
        auto where = std::ranges::find_if(prices,
            [first] (double price) ❸
            {
                return price > first;
            }
        );
        if(where != prices.end()) ❹
        {
            return *where - first; ❺
        }
        else
        {
            return 0.0; ❻
        }
    }
}
```

- ❶ Throws an exception for empty prices
- ❷ Gets the first price
- ❸ Captures `first` by value for use in the lambda
- ❹ Checks that you're not at the end, because `end()` means nothing was found
- ❺ Returns the difference between the value at the found position and the first value
- ❻ Returns `0.0` to indicate that no profit is possible

Your new analysis function, `profit_on_first_uptick`, takes a `std::vector`, as does the average function. Using `valid_prices_as_vector`, you can call the new function in `main`. Add a call at the end of `main` to report the potential profit:

```
double potential_profit =
    stock_prices::profit_on_first_uptick(valid_prices_as_vector);
std::cout << "Potential profit " << potential_profit << '\n';
```

Build and run your program. A sample run might look like this:

```
Please enter some numbers.
>1.25
>0.75
>bye
Got 2 price(s)
The following are valid:
1.25
0.75
with average 1
Potential profit 0
```

Did you actually find a price higher than the first price? In this case, no: you hit the end of the vector without finding a higher price. You may not have made a profit, but you have now written quite a large program. Well done.

Understanding Lambdas and Views in More Depth

You've used algorithms a few times, and now you know how to write a lambda and use a view, but there's a bit more you should know about lambdas and views before you move on.

Lambda Captures by Value

You've written a few lambdas, and you've even used the capture `[]`, also called a *capture group*. You've captured only one variable by value so far. Let's explore what else is possible.

Suppose you hold your nerve and decide to wait for a minimum profit, rather than selling as soon as the price rises above your initial investment. The lambda in `profit_on_first_uptick` checks for a price greater than the first. You can check for a difference greater than a required profit instead:

```
(price - first) >= required_profit;
```

This tells you if you could have made the required profit from the prices.

Declare a new function in *analysis.h*, taking the prices along with a `required_profit`:

```
bool required_profit_possible(const std::vector<double> & prices,
                             double required_profit);
```

To implement the function, you'll need to know how to capture the new value in the lambda. Previously, you put one variable in the capture group:

```
[first] (double price) ❶
{
    return price > first;
}
```

❶ Captures one variable by value in the capture group `[]` and thus copies it

Now you need to capture another value. Add `required_profit` to the group, using a comma to separate it from `first`:

```
[first, required_profit] (double price)
{
    return (price - first) >= required_profit;
}
```

Pulling this together gives you the function shown in [Example 6-5](#).

Example 6-5. A new function in `analysis.cpp` that captures two variables by value

```
bool required_profit_possible(const std::vector<double> & prices,
                             double required_profit)
{
    const double first = prices.front();
    auto where = std::ranges::find_if(prices,
        [first, required_profit] (double price) ❶
        {
            return (price - first) >= required_profit;
        }
    );
    return where != prices.end(); ❷
}
```

❶ Captures two variables by value

❷ Checks for a suitable value

Pick a `required_profit` and call this function from `main`. Here's the whole listing:

```
#include <algorithm>
#include <iostream>
#include <ranges>
#include <vector>

#include "analysis.h"
#include "input.h"

int main()
{
```

```

stock_prices::test_input();

std::cout << "Please enter some numbers.\n";
auto prompt = [] () { std::cout << '>'; };
auto prices = stock_prices::get_prices(std::cin, prompt);
std::cout << "Got " << prices.size() << " price(s) \n";

std::cout << "The following are valid:\n";
auto valid_prices = std::views::filter(prices,
    [](double p) { return p >= 0.0; } );
for(double price : valid_prices)
{
    std::cout << price << '\n';
}
const std::vector<double> valid_prices_as_vector =
    std::ranges::to<std::vector>(valid_prices);
const double mean = stock_prices::average(
    valid_prices_as_vector
);
std::cout << "with average " << mean << '\n';
double potential_profit =
    stock_prices::profit_on_first_uptick(valid_prices_as_vector);
std::cout << "Potential profit " << potential_profit << '\n';
const double required_profit = 1.75; ❶
bool possible =
    stock_prices::required_profit_possible(valid_prices_as_vector,
        required_profit); ❷
std::cout << "Required profit possible " << possible << '\n'; ❸
}

```

- ❶ Picks a required profit
- ❷ Finds out if this profit is possible
- ❸ Displays the result

When you build and run the code now, you get an indication of whether the required profit is possible:

```

Please enter some numbers.
>1.25
>2.13
>4.51
>bye
Got 3 price(s)
The following are valid:
1.25
2.13
4.51
with average 2.63

```

```
Potential profit 0.88
Required profit possible 1 ❶
```

❶ Indicates if the profit is possible

Annoyingly, `std::cout` prints 1 for true and 0 for false. To display true or false instead, you can use `std::boolalpha` from the `<ios>` header before you stream out the bool, like this:

```
std::cout << "Required profit possible " << std::boolalpha << possible << '\n';
```



`boolalpha` is a *manipulator* for a stream: it manipulates a bool, showing it as true or false rather than 0 or 1. Manipulators provide options for controlling how characters are used in input and output streams, for example, changing how many digits are shown after a decimal point. In [Chapter 9](#), you will go back to using `std::println`, which directly prints true or false for a bool, and provides other ways to format output, so you won't need manipulators.

Now, you might want to capture several more values. Two isn't so many, but more will give a long list. You can use an equal sign instead of a list to indicate that you want to capture any variable used by value:

```
[=] (double price)
{
    return (price - first) >= required_profit;
}
```



If you find yourself needing a lot of captured variables in a lambda, that might be a sign that you're trying to do too much at once. Use `=` sparingly to avoid capturing something by mistake. Explicit is often better than implicit.

Now, you can't change or mutate variables you've captured by value. To change them, add the word `mutable` after the parameters:

```
[first, required_profit] (double price) mutable ❶
{
    first += 42.0; ❷
    return (price - first) >= required_profit;
}
```

❶ Says the lambda might change or mutate a captured variable

❷ Changes `first` (which is a silly idea here!)

If you want to change a captured variable, you have a clearer option: you can capture values by reference instead.

Lambda Captures by Reference

Changing the value of `first` is contrived, and not very sensible, but it does illustrate what is possible.

To take a capture by reference, add the reference symbol `&` to that variable:

```
[&first, required_profit] (double price) ❶
{
    first += 42.0; ❷
    return (price - first) >= required_profit;
}
```

❶ Captures `first` by reference and `required_profit` by value

❷ Changes `first` (a silly idea here, too!)

Since `first` is `const`, you will get a compiler error along the lines of:

```
assignment of read-only reference 'first'
```

This is just another reason why it's sensible to mark variables `const` when you don't intend to change them.

If you change the declaration to be non-`const`, your new code will compile:

```
double first = prices.front();
```

You've learned how to use `[=]` to capture anything needed by value, and you've seen a mixture of captures by reference and by value. If you want to capture *everything* you need by reference, use `[&]`.

Beware dangling references

If you capture variables by reference and the lambda outlives the referenced variable, you can get into trouble. This is called a *dangling reference*. If you capture a variable by reference, it might go out of scope. Here's an example from [ACCU talk "Let's Look at Lambdas," by Roger Orr](#):

```
#include <functional>
#include <iostream>

std::function<int(int)> make_adder(int value)
{ ❶
    return [&value](int n) { return n + value; }; ❷
} ❸
```



```

int main()
{
    auto add_ten = make_adder(10); ❹
    int result = add_ten(9); ❺
    std::cout << result; ❻
}

```

- ❶ value in scope
- ❷ Uses value by reference
- ❸ value goes out of scope
- ❹ Gets a `std::function` using value by reference, which has gone out of scope
- ❺ Calls the `std::function`
- ❻ Outputs something, but `result` provokes undefined behavior

Using Clang on [Godbolt](#) outputs 18 rather than 19.

Put the capture group back to `[first, required_profit]`, and then make `first` `const` again, as you had it before—while changing `first` demonstrates what’s possible, it’s not necessarily sensible for this project.

Some best practices for lambdas and capturing are:

- Capture by value, unless it is too expensive.
- Capture explicitly, and try to avoid `[=]` and `[&]`.
- Try to avoid mutable lambdas.

Composing Views

It’s easy to compose views together. Create a new file called *views_experiment.cpp*, and let’s find out how. You’ll hardcode a vector of values this time, to save you from typing in more numbers. Again, you will filter out negative prices and then show prices that are cheaper than a required price.

Up to now, you’ve copied views into a vector for analysis. Now you’ll use the view directly.

You can use another view called `take_while` to take prices while they are lower than the required price. `take_while`, like `filter`, is a range adaptor. *Range adaptors* provide a view of the underlying data, filtering it by the criteria you specify or transforming it by a function. There are many other range adaptors, including `take` (which

takes a specific number of elements), `skip` (which ignores a number of elements), and `skip_while` (which ignores elements that match a predicate). If you want to experiment further, [CppReference](#) gives a full list. Figure 6-2 shows the view that results if your `take_while` elements are squares.

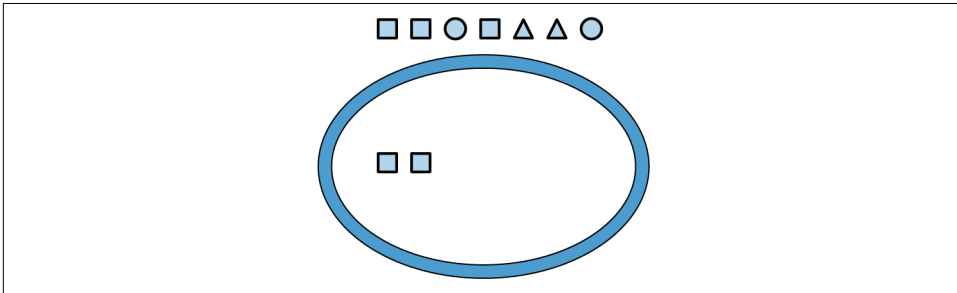


Figure 6-2. A view taking elements while they are squares. Notice it takes the first two squares and then stops as soon as it gets a circle (or anything that isn't a square)

You can create a view that takes initial prices lower than the required value and then print the views to demonstrate that the `take_while` view has no effect on the original filtered view:

```
#include <iostream>
#include <ranges>
#include <vector>

int main()
{
    const std::vector prices{3.76, 1.5, -1.0, 3.0, 4.0, -2.0, 99.4}; ❶
    const double required_price{4.75}; ❷
    auto non_negative = [](double price) { return price >= 0.0; }; ❸
    auto too_cheap = [required_price](double x)
    {
        return x <= required_price;
    }; ❹
    auto valid_prices = std::views::filter(prices, non_negative); ❺
    auto no_good = std::views::take_while(valid_prices, too_cheap); ❻
    std::cout << "Too cheap:\n"; ❼
    for(double p : no_good) ❼
    {
        std::cout << p << '\n';
    }
    std::cout << "Valid prices:\n"; ❽
    for(double p : valid_prices) ❽
    {
        std::cout << p << '\n';
    }
}
```

- ❶ Creates a vector of prices
- ❷ Defines a required price
- ❸ Creates a lambda to filter out negatives
- ❹ Creates a lambda to take prices while they are less than the required price
- ❺ Creates a view filtering out negatives
- ❻ Takes a view of prices lower than required
- ❼ Displays the prices that are too cheap
- ❽ Shows the first view is unaltered

Build this single `main` file. You aren't using any other source files this time, so you only need to use `views_experiment.cpp` in the instructions. The output shows the prices that are too cheap and then shows that your first view is unchanged:

```
Too cheap:
3.76
1.5
3
4
Valid prices:
3.76
1.5
3
4
99.4
```

You can actually compose the two views together in one line of code. There are two ways to create a view. So far, your parameters have been the container or view, and a predicate:

```
auto valid_prices = std::views::filter(prices, non_negative);
```

An alternative starts an expression with the prices and uses the pipe operator, `|`, to send this expression to a view with a predicate:

```
auto valid_prices = prices | std::views::filter(non_negative);
```

Now, `valid_prices` is another view, so you could use another pipe to obtain the second view:

```
auto valid_prices = prices | std::views::filter(non_negative);
auto no_good = valid_prices | std::views::take_while(too_cheap);
```

You might start running out of ideas for clear names at this rate. You can chain or compose these two views together in one statement:

```
auto no_good = prices | std::views::filter(non_negative) ❶  
                    | std::views::take_while(too_cheap); ❷
```

❶ Filters negative prices

❷ Takes prices while they are too cheap

This might not seem all that useful for a couple of views, but chaining *lots* of views together can make your code clearer.

Swap your view experiment to use the pipe:

```
#include <iostream>  
#include <ranges>  
#include <vector>  
  
int main()  
{  
    const std::vector prices{3.76, 1.5, -1.0, 3.0, 4.0, -2.0, 99.4};  
    const double required_price = 4.75;  
    auto non_negative = [](double price) { return price >= 0.0; };  
    auto too_cheap = [required_price](double x) { return x <= required_price; };  
    auto no_good = prices | std::views::filter(non_negative)  
                      | std::views::take_while(too_cheap);  
    std::cout << "Too cheap:\n";  
    for(double p : no_good)  
    {  
        std::cout << p << '\n';  
    }  
}
```

When you build and run your program, you'll see which prices are too cheap:

```
Too cheap:  
3.76  
1.5  
3  
4
```

As before, the prices were 3.76, 1.5, -1.0, 3.0, 4.0, -2.0, 99.4, and you required 4.75. Once the negative prices are filtered, all but the last value, 99.4, are too cheap.

There's one last important detail about views that might not be immediately obvious. Let's have a look.

Lazy Views

When you copy data into a vector, *all* the data gets copied, regardless of whether you want to view it. In contrast, a view doesn't do anything until you use it. This is called *lazy evaluation*, because the view is evaluated only when asked.

When you created a view, you chained a filter and take_while together:

```
auto no_good = prices | std::views::filter(non_negative)
                    | std::views::take_while(too_cheap);
```

However, the filtering and taking don't happen yet. You can prove this by adding a `std::cout` call in the second predicate:

```
#include <iostream>
#include <ranges>
#include <vector>

int main()
{
    const std::vector prices{3.76, 1.5, -1.0, 3.0, 4.0, -2.0, 99.4};
    const double required_price = 4.75;
    auto non_negative = [](double price) { return price >= 0.0; };
    auto too_cheap = [required_price](double x)
    {
        std::cout << "Comparing " << x << '\n'; ❶
        return x <= required_price;
    };
    auto no_good = prices | std::views::filter(non_negative)
                    | std::views::take_while(too_cheap); ❷
    std::cout << "Too cheap:\n"; ❸
    for(double p : no_good) ❹
    {
        std::cout << p << '\n';
    }
}
```

- ❶ Adds output when called
- ❷ Creates a view but doesn't call the predicates yet
- ❸ Displays a message, as before, after the view is created
- ❹ Iterates the view, so it calls the predicates now

When you build and run your code this time, you will see the predicate's message after the Too cheap: output:

```
Too cheap: ❶
Comparing 3.76 ❷
3.76
```

```
Comparing 1.5
1.5
Comparing 3
3
Comparing 4
4
Comparing 99.4
```

- ❶ Displays a message after the view is made
- ❷ Shows that the predicate is called when the view is used

Lazy evaluation can make code more efficient.



Don't be shy about adding output to your code to see what's happening when. A well-known computer scientist, Brian Kernighan, once said, "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."

Conclusion

You used lambdas and views from the `ranges` library in this chapter. Views let you pick certain elements from a range. Many use predicates, which can be named functions, like your `negative` function, or anonymous functions, known as lambdas. Lambdas look like named functions, but don't state the return type, and they start with `[]` (known as the *capture*). You used your lambda to prompt for numbers:

```
[] () { std::cout << '\n'; }
```

You learned how to capture variables for a lambda, by reference and by value—two important ideas you have met before and will use again and again:

- Using `[=]` captures everything you need by value.
- You cannot change by-value captures, but you can add `mutable` if you want.
- Using `[&]` captures everything you need by reference.
- You can name specific variables to capture, like `[first]`.
- You can mix by-reference and by-value captures, like `[&first, required_profit]`.

You used `std::function` to take a prompt so that you could use a `no_op` lambda in tests: `[](){}.`

The prompt function you used was `<void()>`, so it takes no parameters and has a void return. You can use named functions as well as lambdas in a `std::function`, provided the signature matches.

You also used lambdas in ranges' views. Views are lazy, meaning they are evaluated on demand. You can compose views using the pipe operator, like this:

```
auto data = prices | std::views::filter(non_negative)
                  | std::views::take_while(too_cheap);
```

You also saw how to convert a view to a vector, using either `std::ranges::to` or the view's `begin` and `end`. This gives you a copy of the elements in the view. You did this to call functions in *analysis.cpp*.

You built a larger program, reusing your existing source files. You now have a simple trading strategy. It's not going to make you rich, but you know so much more C++ now.

I suspect you might be bored with typing in made-up prices by now, so in [Chapter 7](#), you'll find out how to make your computer generate those fictitious prices for you using random numbers.

Random Numbers

In [Chapter 6](#), you improved `get_prices` so you could vary the prompt using a lambda, and you put this more general version in `input.cpp`. In this chapter, you will write an overload of `get_prices` using “random” numbers and use it to build a small trading app that allows you to sell fictitious stock. Using randomly generated prices means you don’t need to type in numbers to run your program.

If you generate prices that behave like stock prices, you can even try some trading strategies and see how much money you lose (or make). You tried one such strategy in [“Using Lambda Captures for Fun and Profit” on page 120](#), finding the profit on the first uptick. I will show you how to generate prices, based on a simplified version of the [Black-Scholes equation](#). This uses a random variable (also called a *stochastic* variable), representing uncertainty. Many financial institutions use stochastic models to investigate what might happen under different circumstances, such as an interest rate rise.



Most programming languages have a way to generate numbers that appear to be arbitrary so that you are unlikely to be able to guess what number comes next. These numbers are often called *pseudo-random*, because they come from a mathematical function. If you know its details, you can work out what comes next. The numbers are therefore not *truly* random but come close enough for games and the like. If you need true randomness, you need to look beyond software—software and code are deterministic, but random numbers are not. Thus, when I use the word *random* in this chapter, I mean *pseudorandom*.

In [Chapter 8](#), you will learn how to load prices from a file, so you will have a few ways to get the prices. Over the rest of the book you will learn how to build a bigger trading app.

Generating Random Numbers

Let's start by generating a random number. Create a new *main.cpp* file. You're going to write a function for an experiment with C++'s random numbers. Include `<iostream>` for output and `<random>` for the random features:

```
#include <iostream>
#include <random>

void random_experiment()
{
}

int main()
{
    random_experiment();
}
```

You can generate random numbers in many ways. Numbers have a numeric type, like `int` or `double`. The `random` library uses templates, so you can state which type you want. Along with the type, you need to specify a *distribution*: that is, how the numbers should be spread out or distributed. The simplest distribution is *uniform*, meaning that each item is equally likely (like fair dice or tossing a coin).

If the dice are unfair or the coin has been manipulated somehow, the distribution would be *weighted*, or biased. [Figure 7-1](#) compares the outcomes of rolling fair and unfair dice 120 times, along with the expected theoretical outcome for a fair die, which is 20 of each number. The biased die is much more likely to roll a 6, as you can see from the highest bar on the right.

Let's start with a uniform distribution. This is called a `std::uniform_int_distribution`. To roll a die, you want a number between 1 and 6, inclusive. You can specify the type in the familiar `<>` brackets and provide lower and upper bounds, like this:

```
std::uniform_int_distribution<int> distribution{1, 6};
```

C++ can work out the numerical type by deducing it from the bounds you provide, so you don't need to specify `<int>`:

```
std::uniform_int_distribution distribution{1, 6};
```

You met this feature, *class template argument deduction* (CTAD), in [Chapter 4](#).

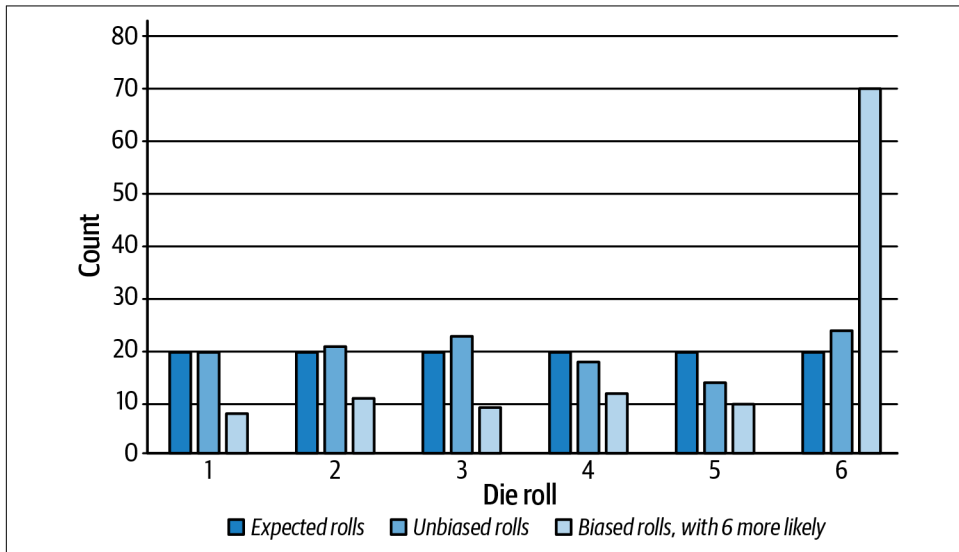


Figure 7-1. A plot of dice rolls, comparing uniform and biased dice

The distribution drives the spread of numbers. Each distribution has an operator, `()`, which takes a *uniform random bit generator*. (You met another *call operator*, `()`, when you learned about function objects in “[Binary Operators and Predicates](#)” on [page 106](#).) A *bit*, of course, is a 0 or 1, and you can write numbers in binary as 0s and 1s. This generator provides numbers made from random bits, such that all possible bit patterns are equally likely.

The distribution then applies a mathematical function to the number from the generator so that, in the long run, your “random” numbers are distributed as required. Generators are sometimes referred to as random-number *engines*, since they drive the distributions. C++ provides several such engines, each with different pros and cons. Some will start repeating the number sequence after a few thousand times; others take longer to repeat but need more state. They are all good enough for games and relatively small simulations. If you want to do cryptography or a large simulation requiring billions of outputs, you may need an external C++ library.



C++ has a `default_random_engine`, which is *implementation defined*, meaning it can vary between toolchains. It often means something called `std::mt19937`, which is a *Mersenne Twister*. This has a lot of state but takes a long time to repeat.

The Mersenne Twister uses a prime number, $2^{19937} - 1$, in a complicated calculation to create a new number from the last few numbers generated. A prime number that is one less than a power of two is called a *Mersenne prime*. The second part of the name, *twister*, is used because the engine swaps bits around before returning a number, as though it is twisting the number.

All the engines have a starting state, which you can influence with a *seed*: a number to start the engine off. If you don't specify a seed, a hardcoded value in your toolchain is used, and you will get the same numbers each time you run your program. If you specify the same seed each time, you will also get the same run of numbers. That can be useful for testing or replicating problems.

To get a different run of numbers each time, which you want for a game or simulation, you need to provide a “random,” or at least varying, seed. You can create a random seed using a `std::random_device`. This is a special generator for seeding other engines, often using the hard-drive state and other physical things from your machine that vary. It is supposed to produce “**nondeterministic random numbers**”.

So you need a seed, an engine, and a distribution. Try these in your new *main.cpp* file:

```
#include <iostream>
#include <random>

void random_experiment()
{
    std::random_device rd{}; ❶
    std::default_random_engine generator(rd()); ❷
    std::uniform_int_distribution distribution{1, 6}; ❸
    const int roll = distribution(generator); ❹
    std::cout << "Dice roll " << roll << '\n'; ❺
}

int main()
{
    random_experiment(); ❻
}
```

- ❶ Makes a `random_device` to generate a seed
- ❷ Makes an engine, using the random seed returned by calling `random_device`
- ❸ Makes a distribution, to provide numbers between 1 and 6 inclusive

- ❹ Gets a random `int` between 1 and 6
- ❺ Displays the number
- ❻ Calls the `random_experiment` function

Build your single *main.cpp* function and run the program a couple of times. You might see different output. In theory, you could get the same output twice, but it's unlikely. You might see something like this:

```
Dice roll 1  
Dice roll 6
```

Let's use random numbers to generate prices. There are several ways to do this. I'll show you two, starting with how to use a `uniform_int_distribution` first.

Writing an Overload for a Function

Given a current price, the next price might be the same, higher, or lower. Rather than a dice roll, you can use -1, 0, or 1 to represent these changes. So you will use a `uniform_int_distribution` from -1 to 1:

```
std::uniform_int_distribution distrib{-1, 1};
```

If you pick a starting price, you can use each random number to increase or decrease it by a fixed percentage (say, 1%) or leave the price as it is.

Open your previous *input.cpp* and *input.h* files. Look at the header file. In [Example 6-2](#), you declared a function to get prices from a stream, providing a variable prompt:

```
std::vector<double> get_prices(std::istream & input_stream,  
                             std::function<void ()> prompt);
```

Let's write another function to get prices. Since this time the prices will be generated randomly, you won't use a stream. You don't need a prompt, either, because the machine is generating the numbers. You can give the new function the same name, `get_prices`, giving a different overload or version. (You met the idea of overloaded functions in [“Understanding `println` and `cout` in Depth](#)” on page 12 and have used several from the standard library—time to write your own.)

First, declare your overloaded `get_prices` function in *input.h*. It will return a vector of doubles, as the previous `get_prices` function did. Since you don't have a stream or prompt as parameters, you need to say how many numbers you want. A `size_t` is suitable. These are unsigned whole numbers, which you met in [Chapter 4](#). You can't

have a negative or fractional number of prices, and using the `size_t` type will prevent these.¹

Add the overload declaration to the *input.h* header:

```
std::vector<double> get_prices(double price, size_t count);
```

Now you need a definition in *input.cpp*. You will return a vector filled with `count` prices. The first will be the given price, and the subsequent values will be (pseudo-)random.

You've declared vectors in a couple of ways so far, using `{}` to provide zero or more values and using `()` to provide a pair of iterators to copy values from (see [“More on Iterators” on page 107](#)). You can also specify how many elements you want initially, giving a specific value:

```
std::vector prices(count, 0.0);
```

You asked for the double `0.0`, so you don't need to specify a type. The default double is `0.0`, so you don't need to spell that out if you want `0.0`, but you *do* need to specify the type in `<>`:

```
std::vector<double> prices(count);
```

You can use `std::ranges::generate` to overwrite elements in a vector. Specify all the elements using `begin` and `end`. The value comes from a generating function.

You are going to generate prices. You can use a lambda to add or subtract a percentage of the previous price, or leave it unchanged. You therefore need to include `<algorithm>` and `<random>`.

Add the new function to *input.cpp*, as shown in [Example 7-1](#).

Example 7-1. A function to simulate prices using a uniform random int

```
#include <algorithm> ❶
// ...
#include <random> ❷
// ...

namespace stock_prices
{
    // ...

    std::vector<double> get_prices(double price, size_t count) ❸
    {
        std::vector<double> prices(count); ❹
```

¹ In general, you use `std::size_t` and include `<cstdint>`, but the `<vector>` header makes this visible.

```

    const double step = price/100.0; ❸

    std::random_device rd{}; ❹
    std::default_random_engine gen(rd()); ❺
    std::uniform_int_distribution distrib{-1, 1}; ❻

    auto next_price = [step, &price, &gen, &distrib]() ❼
    {
        price += step*distrib(gen);
        return price;
    };
    std::ranges::generate(prices.begin(), prices.end(), ❽
        next_price); ❾
    return prices;
}
}

```

- ❶ Includes algorithms for ranges' generate
- ❷ Includes random for random numbers
- ❸ Defines an overloaded get_prices function
- ❹ Makes a vector of count, each with value 0.0
- ❺ Uses a step of 1% (0.01) of the original price
- ❻ Sets up a random generator and distribution
- ❼ Declares a lambda taking step by value and price, gen, and distrib by reference
- ❽ Overwrites from prices.begin() until prices.end(), using the lambda to obtain values

Call the existing tests and your new function from main:

```

#include <iostream>

#include "input.h" ❶

int main()
{
    random_experiment();
    stock_prices::test_input(); ❷
    const auto prices = stock_prices::get_prices(100.0, 10); ❸
    std::cout << "Got prices:\n";
    for(double price: prices) ❹

```

```

    {
        std::cout << price << '\n';
    }
}

```

- ❶ Includes your header to get prices
- ❷ Calls the existing test function
- ❸ Requests 10 prices from the new overloaded function
- ❹ Displays the prices

Don't forget to add *input.cpp* to your build.

Try running your code a few times and see what prices you get. Table 7-1 shows prices from three runs, including the starting prices of \$100. You will get 10 simulated prices, but your values will probably be different. At the moment, this value is returned as 100 rather than \$100.00—you'll learn about formatting output in Chapter 9.

Table 7-1. Table of simulated stock prices

1st	2nd	3rd
100	100	100
100	99	101
99	98	100
100	99	99
100	99	98
99	98	98
99	98	97
100	97	98
100	98	99
101	99	98
102	100	97

Figure 7-2 plots these prices, showing how they might go up or down over time, even though they all started from \$100.

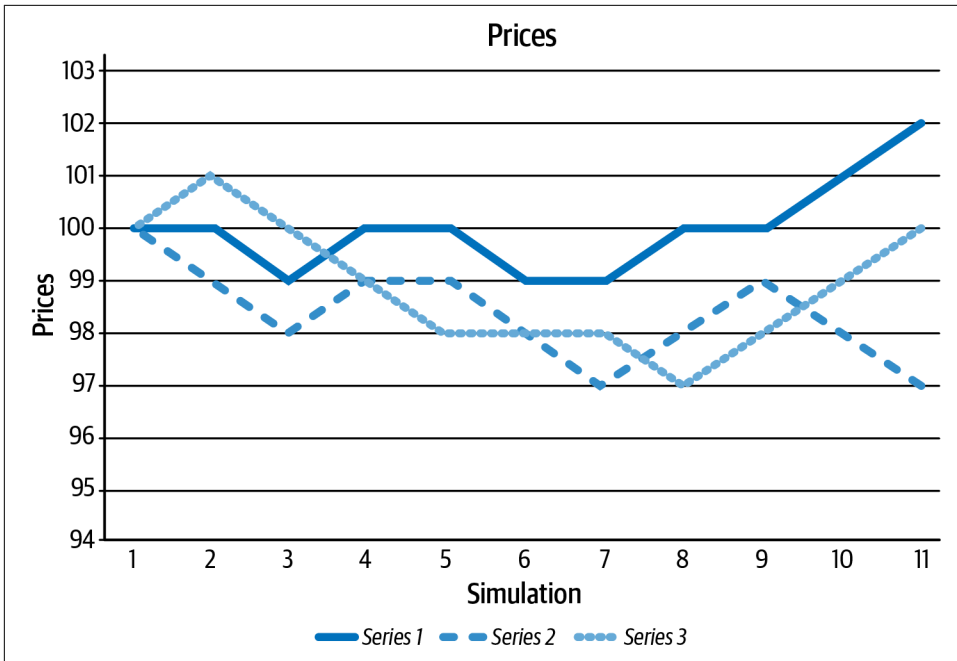


Figure 7-2. A plot of simulated prices

Building a Trading Game

If you include your *analysis.h* header in *main.cpp*, you can try your `profit_on_first_uptick` function in *main*. Add the call after you stream out the prices:

```
const auto profit = stock_prices::profit_on_first_uptick(prices);
std::cout << "Profit " << profit << '\n';
```

Don't forget to add *analysis.cpp* to your build. You might even see a profit:

```
Got prices:
99
100
101
102
101
101
101
100
101
102
Profit 1
```


The first price is \$99, so the second price of \$100 is the first uptick, giving a \$1 profit—not the best possible profit! Since the values can go down as well as up, you might not make a profit at all or you might make a larger profit. An increase of 1% is as likely as a decrease, since you used a uniform distribution. But you will learn more C++!

The trading simulation program will use a starting price of \$100.00 and then get the simulated prices and display them one at a time. The game stops at the end of the prices or when you sell. You can sell your stock by pressing *s* followed by Enter. If you type any other character, it will move to the next price. If you sell, it calculates your profit.

Write a new function called `trading_game` in your *main.cpp* file, above the `main` function. Now you need to input a single character. You’ve used characters in messages before, like `'\n'`. To declare a single character, you use the type `char`, for example:

```
char character{'\n'};
```

You will learn more about chars in [Chapter 9](#).

Loop over the prices, as shown in [Example 7-2](#), and see if you (or a friend who plays your game) want to sell or not.

Example 7-2. A trading game

```
void trading_game()
{
    const double start_price = 100.0; ❶
    std::cout << "Stock bought for: " << start_price << '\n'; ❶
    auto prices = stock_prices::get_prices(start_price, 10); ❷
    for(auto price : prices) ❸
    {
        std::cout << "Current price: " << price << '\n';
        std::cout << "Press (s) to sell\n>";
        char choice{}; ❹
        std::cin >> choice; ❺
        if (choice == 's') ❻
        {
            const double profit = price - start_price; ❼
            std::cout << "Profit " << profit << '\n'; ❼
            break; ❽
        }
    }
    std::cout << "Game over\n"; ❾
}
```

- ❶ Uses and displays the `start_price` of \$100.00
- ❷ Gets random prices
- ❸ Loops over the prices
- ❹ Declares a `char` to hold a single character
- ❺ Streams in a character
- ❻ Compares the character with `s`, meaning sell
- ❼ Calculates and displays the profit
- ❽ Breaks out of the loop
- ❾ Indicates the game is over

You have seen almost everything in this example before, but there is one new C++ keyword here: `break`. Using `break` tells the program to halt a loop and go to the next line of code (after the loop's closing brace). Call the new function from `main` and build your code.

To play the game, press `s` followed by Enter to sell, or any other character followed by Enter to keep going:

```
Stock bought for: 100
Current price: 101
Press (s) to sell
>a
Current price: 102
Press (s) to sell
>a
Current price: 103
Press (s) to sell
>s
Profit 3
Game over
```

Well done. You now have a small game to play.

There are other ways to generate prices, so in the next section, let's try using a very different distribution.

Understanding Code with Random Numbers (and Vectors) in Depth

The `get_prices` function overload you wrote generates prices randomly, using a fixed percentage to move up or down. You used the `uniform_int_distribution` to get `-1`, `0`, or `+1`, giving the lower and upper bounds:

```
std::uniform_int_distribution distrib{-1, +1};
```

The type `int` is deduced from the parameters using CTAD. You can also make a distribution for unsigned numbers, which you met in [Chapter 4](#). For example, `0u` is an unsigned `int`. When you create a distribution for your first `random_experiment`, you could use an unsigned `int` instead, like this:

```
std::uniform_int_distribution distrib{1u, 6u};
```

There are several different numeric types in C++. Although `uniform_int_distribution` has `int` in the middle, in this context, the `int` means integral or whole numbers, not integers. You could also use a `short`, which covers a smaller range of numbers, or a `long`, like `1l`; or even a `long long`, like `1ll`, which covers a larger range. A `long` can fit at least as many numbers as an `int`, and a `long long` can fit more than a `long`. You can also have unsigned `long`, `1ul`, and unsigned `long long`, `1ull`.

If you want doubles in a range, use the `std::uniform_real_distribution`:

```
std::uniform_real_distribution distrib{1.0, 6.0};
```

Real numbers include whole numbers, negative numbers, fractions, and special numbers like π .

C++ has a couple of other types for real numbers: `float` and `long double`. A `float` takes up less space than a `double` and so can't represent as many numbers. There's a limit on which numbers can be precisely represented for each floating-point type in C++ (or, in fact, any programming language).

To use a `float`, add an *f* to the end of your number:

```
float number = 12.5f;
```

```
long double number = 12.5l;
```

In contrast, a `long double` needs more space but can represent more numbers.



The suffix (letter after the number) is case insensitive, so you can use `u` or `U`, `f` or `F`, and `l` or `L`.

Try some sums with some floating-point numbers:

```
auto as_float = 0.1f + 0.2f;
auto as_double = 0.1 + 0.2;
std::cout << as_float << '\n';
std::cout << as_double << '\n';
```

The call to `std::cout` rounds the results, so you see two values of 0.3:

```
0.3
0.3
```

You can use manipulators, from the `<iomanip>` header, to show greater precision. You set the precision to 18 (see [C++Reference](#) for details), like this:

```
std::cout << std::setprecision(18); ❶
std::cout << as_float << '\n';
std::cout << as_double << '\n';
```

❶ Asks for more decimal places

When you run the code, you see two different values:

```
0.300000011920928955 ❶
0.300000000000000044 ❷
```

❶ Shows the value as a float

❷ Shows the value as a double

Not all real numbers can be represented exactly in a float, double, or long double. A paper by David Goldberg from 1991, called “[What Every Computer Scientist Should Know About Floating-Point Arithmetic](#)”, goes into details, showing how real numbers are represented and why you might see rounding errors. Be reassured that the first call to `std::cout` showed you 0.3. You can control the precision of output, and there are known ways to deal with potential issues, which the Goldberg paper goes into.

Using a Normal Distribution

C++ has several different random-number distributions, including the *normal* distribution, sometimes called a [Gaussian distribution](#). The numbers generated tend to be closer to the mean, and extreme values are less likely. Recall that the *mean* is the total of a collection of numbers divided by their count. People’s heights are often cited as an example of a normal distribution.

The distribution uses the mean and a second statistic, called the *standard deviation*, to control how far the numbers are likely to spread out. [Figure 7-1](#) showed counts of

dice rolls. If you generate normally distributed random numbers instead and count how many you get in ranges, you will see something more like [Figure 7-3](#).

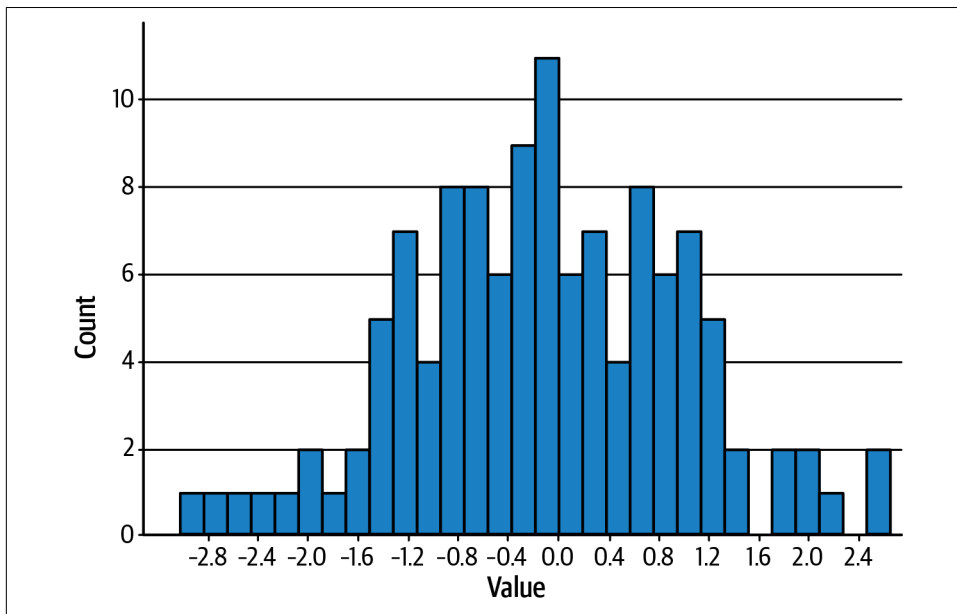


Figure 7-3. A plot of normally distributed numbers

[Figure 7-3](#) shows numbers with a mean of 0.0 and a standard deviation of 1.0, called the *standard* normal distribution. These are the default values for the normal distribution in C++, so you can create one like this:

```
std::normal_distribution normal_dist;
```

You can spell out the mean and standard deviation, too:

```
std::normal_distribution normal_dist{0.0, 1.0};
```

The uniform distributions also used two numbers: a lower bound and an upper bound. The normal distribution's parameters, by contrast, mean something very different. The smallest number represented in [Figure 7-3](#) is -3.01841, and the largest is 2.65152. (You might get different values on another run.) Most of these numbers are near the mean, and the standard deviation controls how far away other numbers may be.

You can use this distribution to generate stock prices in a different way. Rather than using a fixed percentage of the price, you can now vary the change with a random number. Positive and negative numbers are equally likely, but you will tend to get numbers close to zero. You can multiply a random number from the normal

distribution by an amount that is known as *volatility* in finance. The larger the volatility, the more the prices change.



People sometimes model stock prices with a **Weiner process**. To see a simplification of this model, watch my “Diffuse your way out of a paper bag” talk on **YouTube**.

Let’s add another `get_prices` function. This overload will use the normal distribution and take a volatility, along with a starting price and count. Add the declaration to *input.h*:

```
std::vector<double> get_prices(double price, size_t count, double volatility);
```

Think of the volatility as an improvement on the fixed step. In the previous `get_number` function, you used 1% of the price. Now you will multiply the volatility by a normal random number to simulate each percentage price change. Add this implementation to *input.cpp*, as shown in **Example 7-3**.

Example 7-3. Another way to simulate stock prices

```
std::vector<double> get_prices(double price, size_t count, double volatility)
{
    std::vector<double> prices(count); ❶

    std::random_device rd{}; ❷
    std::default_random_engine gen{rd()}; ❷
    std::normal_distribution distrib; ❸

    auto next_price = [volatility, &price, &gen, &distrib]()
    {
        double percent = volatility * distrib(gen); ❹
        price += price * percent; ❺
        return price;
    };
    std::ranges::generate(prices.begin(), prices.end(), next_price); ❻
    return prices;
}
```

- ❶ Creates a vector of count doubles
- ❷ Creates an engine to generate random numbers
- ❸ Creates a standard normal distribution
- ❹ Picks an increment (a fixed amount to increase by) by which to change the price

- 5 Creates a lambda to add the increment times the price to the previous price
- 6 Uses the lambda to generate prices

You can generate prices in the new way from `main`. Try a small volatility, like 0.05:

```
const auto prices = stock_prices::get_prices(100.0, 10, 0.05);
```

Display these, and you will see a greater variety of prices:

```
Got prices:
101.575
102.215
96.4626
96.4578
91.9411
96.3343
96.8093
97.9929
104.387
99.2261
```

The specific values will vary with each run. If you increase the volatility, the prices can get further from the original price and might even go negative.

Considerations for Code That Uses Random Numbers

You have written a couple of functions to simulate prices that take a starting price and required count. You can test them to make sure you get the count you expect. Keeping the generating function separate means you can use a known set of numbers to test code that uses the random numbers.

If you record the seed your engine uses, you can reuse the same seed to regenerate the same “random” numbers for testing.

First, I’m going to show you an option for using a specific seed. I don’t think this is the best approach, but some people do it, so it’s worth knowing. You will also learn about another feature of C++. After that, I’ll show you a better way to structure code that uses random numbers.

As it stands, your functions use a `random_device` directly to get a seed. You can’t affect this from calling code. You *could* change the function signature to take a *default* parameter:

```
std::vector<double> get_prices(double price, size_t count,
    double volatility,
    unsigned int seed = std::random_device{}());
```

The seed uses a default, indicated by the `=` sign, which creates the instance `std::random_device{}()` and then calls it using `()`.

You can add `=` and a value to any parameters at the end of a function declaration. Once you default one parameter, the subsequent parameters need defaults as well. For example, you can make the last parameter a default, like this:

```
void function_with_default(int first, int second = 0);
```

However, you can't provide a default for the first parameter but not the second:

```
void invalid_function_with_default(int first=0, int second);
```

Calling code can either specify the value or omit it (to use the default). You also need to include `<random>` in the header—otherwise the `std::random_device` used in the declaration would be unknown.

You then need to change `get_prices` to use the provided seed. Previously, in [Example 7-3](#), you used a seed from `random_device` in the function:

```
std::random_device rd{};
std::default_random_engine gen{rd()};
```

Change the code to use the provided seed instead:

```
std::default_random_engine gen{seed};
```

Calling code can provide a seed or accept the default:

```
auto seed = std::random_device{}();
std::cout << "Seed " << seed << '\n';
const auto prices = stock_prices::get_prices(100.0, 10, 0.05, seed); ❶
const auto different_prices = stock_prices::get_prices(100.0, 10, 0.05); ❷
```

❶ Specifies a seed

❷ Uses the default seed

Using default parameters can be helpful but does have drawbacks. For instance, you might need to include further headers, even if the default parameter is never used. In a large codebase, extra includes make the build slower. Furthermore, the calling code doesn't make it obvious that a default is being used. You could always use an overloaded function instead and pass the extra parameter, like this:

```
void function_with_two_parameters(int first, int second);

void function_without_default(int first)
{
    function_with_two_parameters(first, 0);
}
```

A single default parameter is fine. If you think you need two or more, try an overloaded function instead.

Alternately, instead of using a default parameter or taking a seed, you can pass in a function to generate numbers and then send in a function that generates a known sequence for testing. Let's see how.

You may have noticed that the two new `get_prices` functions are similar, with several identical lines, as shown in [Example 7-4](#).

Example 7-4. Similarities between the new functions to get prices (this code won't compile)

```
std::vector<double> get_prices(double price, size_t count, double volatility)
{
    std::vector<double> prices(count);

    std::random_device rd{}; ❶
    std::default_random_engine gen{rd{}}; ❶
    some_distribution distrib; ❶

    auto next_price = some_lambda; ❷

    std::ranges::generate(prices.begin(), prices.end(), next_price);
    return prices;
}
```

- ❶ The distributions used vary, but both use a default engine and seed from `random_device`
- ❷ The lambdas used vary

To vary the behavior, you can send a lambda into a single function instead.

Add another overload declaration to *input.h*. You need the first price and a count, along with a `std::function`:

```
std::vector<double> get_prices(double price, size_t count,
                               std::function<double()> next_price); ❶
```

- ❶ Takes a function, such as a lambda, to generate prices

Put the implementation in *input.cpp*:

```
std::vector<double> get_prices(double price, size_t count,
                               std::function<double()> next_price)
{
    std::vector<double> prices(count);
    std::ranges::generate(prices.begin(), prices.end(), next_price);
    return prices;
}
```

You don't need the random setup or the lambda from [Example 7-4](#), since you can put them in a lambda in the calling code. So, in `main`, you can use the normal distribution:

```
std::random_device rd{}; ❶
std::default_random_engine gen{rd()}; ❶
std::normal_distribution distrib; ❶
double price = 100.0;
const double volatility = 0.05;
auto next_price = [volatility, &price, &gen, &distrib]() ❷
{
    double percent = volatility * distrib(gen);
    price += price * percent;
    return price;
};

const auto more_prices = stock_prices::get_prices(100.0, 10, next_price); ❸
```

- ❶ Sets up engine and distribution
- ❷ Captures the variables required to make a new price in a lambda
- ❸ Calls the new `get_prices` function

You can now use the same engine to generate prices in a different way. For example, you can use a `uniform_int_distribution` again:

```
const double step = price/100.0;
std::uniform_int_distribution<> uniform_distrib{-1, +1};
auto next_uniform_price = [step, &price, &gen, &uniform_distrib]() ❶
{
    price += step*uniform_distrib(gen); ❶
    return price;
};
const auto even_more_prices = stock_prices::get_prices(100.0, 10,
                                                         next_uniform_price);
```

- ❶ Uses the same engine

Now you can reuse a single engine, which can be quicker than making several engines. Some people regard the Mersenne Twister as large and relatively slow compared to other engines. C++26 introduced a new engine called `std::philox_engine`, which is smaller and quicker; however, no compilers support this yet at the time of writing. CppReference provides a list of [support by feature for common toolchains](#).²

² CppReference hasn't updated for a while (at the time of writing), so check <https://cppstat.org> as well.



C++ has had a new standard every three years since 2011. C++11 was a big change compared to the previous standards. Since then, various new features have been introduced in 2014, 2017, 2020, and 2023. The code in this book uses C++23, but I'll point out newer features from time to time. If your compiler supports a `-std=c++26` flag, try it.

You won't notice the difference in performance for the code in this chapter, because you have generated only 10 numbers each time. If you needed to do a huge simulation quickly, though, the difference might be noticeable.

Another advantage of the overload using a `std::function` is that you can vary what generates prices. That means you can write tests without random numbers, which makes life simpler.

Creating and Filling Vectors

You have used `std::vector` several times now, and you've seen several ways to create a vector and provide values. You can:

- Provide specific values in an initializer list, using `{}`
- Request a count of a specific value or default
- Create an empty vector
- `push_back` elements
- Copy elements from another vector

Here are some examples:

```
std::vector first_vector{1, 3, -5}; ❶
std::vector second_vector(10, 1.1); ❷
std::vector<double> third_vector(10); ❸
std::vector<double> fourth_vector{}; ❹
fourth_vector.push_back(10.1); ❺
std::vector fifth_vector(first_vector.begin(), first_vector.end()); ❻
```

- ❶ Creates a vector with an initializer list of three ints, 1, 3, and -5
- ❷ Creates a vector of double, with 10 values of 1.1
- ❸ Creates a vector of double, with 10 default values, 0.0
- ❹ Creates a vector of double with an empty initializer list
- ❺ Puts 10.1 in the fourth vector

❸ Creates a vector using the first vector

In “What Happens When You Add to a Vector” on page 77, you saw how the vector’s capacity changes as you add new elements. Starting with the values you need is more sensible than adding elements one at a time, since you avoid having to resize the vector several times.

In that chapter, you used `std::ranges::generate` to put prices in a `std::vector`. Here’s a reminder of the code:

```
std::vector<double> prices(count);
std::ranges::generate(prices.begin(), prices.end(), next_price);
```

First you make a vector with a count of a specific value. You then change the value of each element. The code therefore goes through the elements twice. That’s not a big deal for 10 numbers. C++ can be very efficient, but you can slow it down by doing something several times when you don’t need to.

You can reserve space for the count you need and then put the prices in using another algorithm called `std::generate_n`. Like `std::generate`, this algorithm takes a function to generate values, along with a count of how many. It’s good to get in the habit of reserving what you need and thinking about when you might be doing something inefficiently.

The `reserve` doesn’t make any new elements: it only makes space for them. You can then add new elements without the vector needing to resize and transfer elements to the new space. You need to tell `std::generate_n` where to put elements, and it needs to create new elements rather than overwrite, as `std::generate` does. You use a convenience function called `std::back_inserter` to do this. It takes your container and returns a special iterator that calls `push_back` for you.

Here’s an improved version of `get_prices` using `generate_n`:

```
std::vector<double> get_prices_improved(double price, size_t count,
                                       std::function<double()> next_price)
{
    std::vector<double> prices{}; ❶
    prices.reserve(count); ❷

    std::ranges::generate_n(std::back_inserter(prices), ❸
                           count, next_price); ❹
    return prices;
}
```

❶ Creates an empty vector

❷ Reserves space for count prices

③ Calls `generate_n` on a back inserter for the vector

④ Puts count new prices in the vector

When you used ranges' `generate`, you started with the `begin` iterator because you wanted to overwrite existing elements. This time, you've reserved space but want to add new elements. You therefore need `generate_n` to `push_back` elements for you, so use the `std::back_inserter`.

Figure 7-4 illustrates the difference between not reserving space (left) and reserving space (right). Without a `reserve`, adding a new element to a vector might mean you need to create more space and transfer the elements. With a `reserve`, however, space is available, so elements have somewhere to go. This way you don't need to create space and transfer over and over.

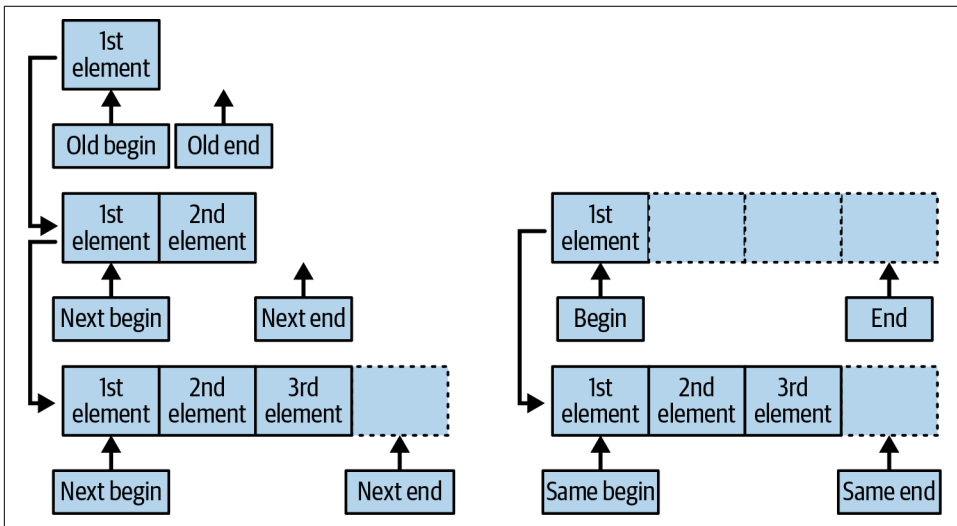


Figure 7-4. Reallocations for each `push_back` versus reserving space up front

Conclusion

This chapter introduced random numbers and taught you more about numeric types in C++. You also had extra practice using a `std::vector`.

You need three things to generate random numbers in C++:

- An engine
- A seed for the engine, though it will default to a fixed value
- A distribution

`std::default_random_engine` is a sensible engine choice for a game or small simulation. The seed controls the numbers generated. Using the same seed will generate the same numbers each time you run your program, though different toolchains might generate different numbers even with the same seed. To pick an arbitrary seed so the numbers change on each run, you can use the `std::random_device`.

You pass the engine to a distribution to get numbers with your required property. You used a few of the random distributions:

- The `uniform_int_distribution`, which is great when you want a whole number from a range, for example, simulating rolling dice. This works for various integral types, including `short`, `long`, and `long long`.
- The `normal_distribution`, which can be used to model stock prices, people's heights, or any situation where you want more numbers to be close to the average. It also works for floating-point types.
- A `uniform_real_distribution` works for `float`, `double`, and `long double`.

The other distributions are useful for more complicated simulations.

You met new ways to create vectors, requesting a count and either a specific value or using the default for your type. You also wrote a function that takes a default parameter. In fact, creating a vector with a count uses a default parameter to give you the two alternatives. Nice.

You wrote several overloads of `get_prices`, one of which took a `std::function`, allowing you to vary how prices are generated. You also learned how to use `char` to represent a single character and `break` to stop a loop.

In [Chapter 8](#), you will discover how to read prices from a file.

Working with Files

You now know quite a lot of C++, but there is more to learn. In this chapter, you will work with files, writing them out and reading them back. You'll generate simulated prices, using the code you wrote in [Chapter 7](#), save them, and learn how to read them back in. You will then have several ways to get prices.

In [Chapter 1](#) you wrote output to a stream, and in [Chapter 2](#) you read input from a stream. Files are also streams in C++, so you know almost everything you need for the project in this chapter. You have also written a function to read a stream, so you've done the hard work already. This chapter will show you how to call your `get_prices` function with a file, rather than with `std::cin`.

I'll also discuss how (and why) to return specific values from `main`, and you will learn about bitwise operators, which are relevant here because a similar idea is used to control files.

Writing to a File

Create a new *main.cpp* file. You will use the `get_prices` function from [Chapter 7](#) to generate prices, so include *input.h*.

Write a function in `main` called `write_to_file` to write the generated prices to a file. You previously used `std::cout` to stream values to the screen. To write to a file in C++, you use an *output file stream*, called `std::ofstream`. This is defined in the `<fstream>` header. File streams are similar to `cout` and `cin`. For example, you use operator `<<` for output.

Try the code in [Example 8-1](#), remembering to add *input.cpp* to your build.

Example 8-1. Writing stock prices to a file

```
#include <fstream> ❶
#include <iostream>
#include <vector>

#include "input.h"

void write_to_file(const std::vector<double> & prices,
                  const std::string & filename)
{
    std::ofstream file{filename}; ❷
    if(file) ❸
    {
        for(auto price: prices) ❹
        {
            file << price << '\n'; ❹
        }
    }
}

int main()
{
    write_to_file(stock_prices::get_prices(100.0, 10, 0.05), "prices.txt"); ❺
}
```

- ❶ Includes file streams
- ❷ Tries to open a file for output
- ❸ Checks that the file opened OK
- ❹ Writes out prices
- ❺ Gets simulated prices and calls `write_to_file` using filename *prices.txt*

You don't need to create `std::cout`: it's available already. To use a file, you need to create a `std::ofstream` object with a filename. When you do that, the file will open, or try to. (You'll see more options for opening a file in [“Understanding Files in Depth” on page 167](#).)

You *can* call `file.close()` to close the file when you are done, but you don't need to. C++ automatically closes the file for you when the file stream goes out of scope, at the closing brace for the `write_to_file` function.

Opening a file might fail, so you need to check that nothing has gone wrong. In [Example 2-4](#), you checked that an input stream was OK using the object in a Boolean context:

```
if(std::cin)
{
}
```

The file is also a stream, so like `std::cin`, you check that your file is OK in the same way:

```
if(file)
{
}
```

The output uses the operator `<<` to stream out values:

```
file << price << '\n';
```

This should be familiar, since you've used this operator with `std::cout` several times now.

Build and run your code, and you will see a *prices.txt* file in the current directory. Open it to see some simulated prices:

```
97.3031
97.1313
94.5203
95.0262
102.107
100.954
95.7085
99.1761
109.866
106.527
```

Detecting and Reporting Problems

At the moment, the program runs, but it doesn't give you any feedback. You can only tell whether it worked by seeing if you have a new *prices.txt* file. For example, when you check whether the file opened OK using `if(file)`, it doesn't report a problem. You could provide an error message for if this happens.

Include `<stdexcept>` so you can throw a `std::runtime_error` if you can't open the file. Add a success message for when the prices are saved so you can tell what your program is doing:

```
#include <fstream>
#include <iostream>
#include <stdexcept> ❶
#include <vector>

#include "input.h"

void write_to_file(const std::vector<double> & prices)
    const std::string & filename)
```

```

{
    std::ofstream file{filename};
    if(file)
    {
        for(auto price: prices)
        {
            file << price << '\n';
        }
        std::cout << "Wrote to prices.txt\n"; ❷
    }
    else
    {
        throw std::runtime_error("Failed to write to prices.txt"); ❸
    }
}

```

- ❶ Includes the standard exceptions header
- ❷ Lets you know prices were written OK
- ❸ Reports a problem

This version of `write_to_file` is better, because it tells you if everything worked or if there is a problem. However, the potential exception will leak outside of `main` if you don't add a `try/catch` block (which you learned about in [Chapter 3](#)). You can add a `try/catch` block to your `main` function and display any problems using the exception's `what` function.

There's another improvement you can make. So far, you have allowed C++ to return an `int` from `main` for you. If you don't provide an explicit `return` statement, C++ returns 0. But you can return an `int` value yourself. By convention, 0 is used to indicate everything worked. Anything other than 0 indicates an error.

If you launch your program with a script, that script can detect the return value and display a message, log the problem, or try to relaunch your program. Change your `main` function to return 1 if there's a problem, like this:

```

int main()
{
    try ❶
    {
        write_to_file(stock_prices::get_prices(100.0, 10, 0.05), "prices.txt");
        return 0; ❷
    }
    catch(const std::exception & e) ❸
    {
        std::cout << e.what() << '\n'; ❹
        return 1; ❺
    }
}

```

```
}  
}
```

- ❶ Tries to write to a file
- ❷ Returns 0 to indicate success (nothing went wrong)
- ❸ Catches an exception
- ❹ Reports the exception's message
- ❺ Returns something other than 0 to indicate failure



You can include `<cstdlib>` to use the values `EXIT_SUCCESS` and `EXIT_FAILURE` rather than using 0 or 1. [CppReference](#) has more details.

Catching any exception in `main` makes the program more user-friendly. Returning a value to indicate a problem means the program can be called from a script, which can detect the problem. It would also be even more user-friendly to give the full path to the file in any messages.

Using the Filesystem Library

C++17 introduced a filesystem library that allows you to work with files and directories. The way files and directories work varies between operating systems. Because of differences like these, before C++17, you had to use nonstandard libraries to work with a filesystem. Reading or writing to a file doesn't require the filesystem library. The filesystem library offers many other features, such as checking file permissions, copying or renaming files, and iterating a directory.

Let's use one feature to report the fully pathed filename whenever the prices are written or if an error occurs.

Include `<filesystem>` in `main`. Then you'll be able to find the current directory, where the *prices.txt* file will be written:

```
const std::filesystem::path path = std::filesystem::current_path();
```

You can add a filename to a path using the operator `/`. This will add a slash if needed:

```
const std::string filename{"prices.txt"};  
const std::filesystem::path path = std::filesystem::current_path();  
const auto fully_pathed_filename = path / filename;
```

You can read this as “path, then filename.” Finding ways to write easy-to-read code is always a good thing.

The `std::filesystem::path` is an object with a `string()` function, which you can use if you want the path as a string. You could also stream the path itself out directly, but the exception message needs to include some extra information to say that the program failed to write to the file. Add the extra details to your `write_to_file` function:

```
#include <filesystem>

void write_to_file(const std::vector<double> & prices,
                  const std::string & filename)
{
    const std::filesystem::path path = std::filesystem::current_path(); ❶
    const auto fully_pathed_filename = path / filename; ❷
    std::ofstream file{filename};
    if(file)
    {
        for(auto price: prices)
        {
            file << price << '\n';
        }
        std::cout << "Wrote to " << fully_pathed_filename << '\n'; ❸
    }
    else
    {
        auto error_message = "Failed to write to " + ❹
            fully_pathed_filename.string(); ❺
        throw std::runtime_error(error_message); ❻
    }
}
```

- ❶ Finds the current path
- ❷ Creates the file with full path for messages
- ❸ Reports where the file was written
- ❹ Creates an error message
- ❺ Uses the `string` function to get the full file path as a string
- ❻ Throws an error with a more informative message

Now you’ve written prices to a file, you’ve revised streams and exceptions, and you’ve even had a taste of the `filesystem` library. Next, let’s see how to read the prices in from a file.

Reading from a File

A `std::ofstream` is an output file stream. To read a file, you can use a `std::ifstream`, which is an input file stream. This is easy to remember: an *output* file stream is called `ofstream`. An *input* file stream is called `ifstream`. The `std::ofstream` is also defined in the `<fstream>` header.

Figure 8-1 shows how the types of streams you’ve met so far are related. The generic types are shown with dashed lines, and the specific types for files and screen output have solid lines. The arrows show which more general type each specific type is based on.

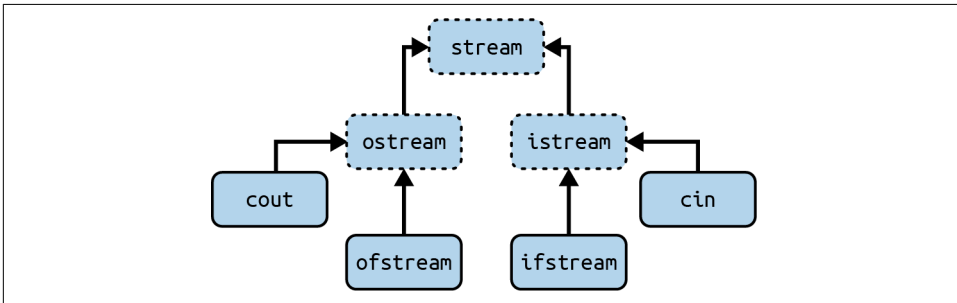


Figure 8-1. The relationships between some of the standard streams

`std::stringstream`, which you met in [Chapter 2](#), is another type of stream that can be used as an `istream` or `ostream`. If a function uses the generic `istream` or `ostream`, you can send it files, `std::cin`, or `std::cout`—as well as the `std::stringstream`, which is flexible and useful for testing. Alternately, you could use a `std::istring` stream for input only and a `std::ostringstream` for output only. Files can also use the more general `std::fstream`, but you need to specify how it should open: for input, output, or both. (We’ll get to that in [“Different File Modes” on page 167](#).)

You can read your *prices.txt* file back using the `get_prices` function you defined in *input.cpp*, using the overload and taking a stream and a prompt. Previously, you used this overload to get prices from the stream `std::cin`. You also included the `<sstream>` header and used the `std::stringstream` in [Chapter 2](#), allowing you to test input and output with streams. An input file stream is also a stream.

The prompt can be a no-op lambda `[](){}` this time, which you may recall is a lambda that captures nothing `[]`, takes no parameters `()`, and does nothing `{}`. As with the `std::ofstream`, the input file stream will open when you declare it and close automatically when it goes out of scope.

Add the code shown in [Example 8-2](#) to *main.cpp*.

Example 8-2. Reading prices from a file

```
void read_from_file(const std::string & filename) ❶
{
    std::ifstream file{filename}; ❷
    if(file) ❸
    {
        auto prices = stock_prices::get_prices(file, [](){}); ❹
        for(auto price: prices) ❺
        {
            std::cout << price << '\n'; ❻
        }
    }
    else
    {
        throw std::runtime_error("Failed to read from file"); ❼
    }
}

int main()
{
    try
    {
        write_to_file(stock_prices::get_prices(100.0, 10, 0.05), "prices.txt"); ❼
        read_from_file("prices.txt"); ❸
        return 0;
    }
    catch(const std::exception & e)
    {
        std::cout << e.what() << '\n';
        return 1;
    }
}
```

- ❶ Defines a new function to read prices from a file
- ❷ Tries to open a file for input
- ❸ Checks the file opened OK
- ❹ Reads prices from the input stream
- ❺ Displays the prices
- ❻ Throws an exception if there is a problem
- ❼ Writes simulated prices, as before
- ❸ Reads the prices in from the file

Build and run your code. You will see some randomly generated prices displayed that will match the values in *prices.txt*. For example, you might see something like this:

```
102.637
102.211
108.415
107.964
105.532
107.052
107.278
110.619
110.053
113.303
```

As before, the prices will vary each time you run your code.

That's almost all you need to know to use files. You did most of the hard work in the first two chapters, when you learned about streams. You could reuse `get_prices` for a stream, because keyboard input and file input both use a stream type. You'll see how to write your own types later in this book. Writing the `get_prices` function in terms of the more general `std::istream` saves you time and trouble.

To recap:

- Working directly with files can be hard to test.
- Using an `istream` or `ostream` instead means you can use `std::cin` or `std::cout` as well as files.
- More importantly, you can write tests using a `std::stringstream` too.

I'll show you some more details on files in the next section.

Understanding Files in Depth

You know how to work with files now, but there are some extra details you should know. Opening a file uses a default *file mode*, which is a number that controls the file's behavior. For example, you can open your file in read-only mode.

Different File Modes

If you run your program twice, it will overwrite the price file, giving you 10 new values every time. Adding new values to an existing file would be more useful. When you open a file for writing, you can ask to *append* output instead, using the `std::ios_base::openmode` called `std::ios::app`. Append mode works whether or not the file already exists. If it does already exist, the new output gets added to the end.

There are several file-opening modes. By default, you get the `std::ios::out` mode for a `std::ofstream`, so the following two lines are equivalent:

```
std::ofstream file{"prices.txt"};
std::ofstream file{"prices.txt", std::ios::out};
```

This mode opens the file for writing and truncates it, so it starts off empty.

If you want to add to an existing file, you'll need to use a different mode. You specify the modes you need after the filename. To specify that you want both out and append, join them together with a pipe character |:

```
std::ios::out | std::ios::app
```

Change the line in [Example 8-1](#) where you create the output file stream so that it uses open and append mode:

```
std::ofstream file{"prices.txt", std::ios::out | std::ios::app};
```

If you run your code now, you will see 10 more prices. The code simulating prices restarts from 100 each time, so you get a bit of a jump after the first 10 numbers:

```
104.583
104.131
103.369
103.218
99.071
95.8083
89.6743
80.6044
78.2807
79.0099
96.6133 ❶
96.0986
90.0869
92.106
91.4286
89.3222
88.1562
93.3136
92.5293
96.9445
```

- ❶ Shows more prices, which have jumped from the previous \$79.0099 by much more than 5%

The simulation is only supposed to move by the given volatility of 5%. More than 5% is possible, but the jump from \$79.01 (approximately) to \$96.61 is more than 20%, which is highly unlikely. This jump happens because you're restarting from the *original* price, rather than the *last* price. You do have more prices now, though.

Let's look at how the pipe character joins the file modes and then revisit the price jump.

Bitwise Operators and Bitmasks

You met the logic operators in [Chapter 2](#), including `&&` for and. You use `||` for or and `!` for not. Now, you used the pipe character `|` to join two modes. The single pipe means something different, operating on bits.

In [Chapter 7](#), you learned that a *bit* is a 0 or 1 and that you can write numbers in binary as 0s and 1s. You can use bits in a number to indicate whether something is on or off—including file modes. Applying bitwise operators to the binary representations of numbers is a common way to combine modes of any kind, not just file opening modes.

The specific values for `std::ios::out` and the other `ios` modes are *implementation defined*, which means your toolchain can decide how to implement the feature but must document the choice. The ISO C++ language standard documents some features as implementation-defined. For example, the way random numbers are produced varies between implementations. If you use the same seed for `clang` and `gcc`, you might still get different numbers. The `filemodes` are implementation-defined. Their exact values don't matter and can vary between toolchains. What's significant is how they are combined and what they do. Let's think through what's going on.

Consider how numbers work in binary: each 0 or 1 represents a power of two. If you have four digits, the positions represent 2^3 (or 8), 2^2 (or 4), 2^1 (or 2), and 2^0 (or 1):

```
0000 = 0
0001 = 1
0010 = 2
0100 = 4
1000 = 8
```

To make three, you need `0001` and `0010`: a one and a two, giving `0011`, a three. You can achieve this using the pipe operator, `|`. Here it performs a bitwise or, for example:

```
0001 | 0010 == 0011
0011 | 0010 == 0011
```

If both inputs have a zero in a column or specific bit, you get zero. `0001` and `0010` start with two 0s, so the result starts with two 0s. If either input has a one in a column, the result has a one in that column, so the previous result ends in 11.

You use the bitwise operator `&` for *and*:

```
0001 & 0010 == 0011
0011 & 0010 == 0010
```

If either input has a zero in a column, the result will have a zero. If both inputs have a one in a column, the result will have a one in that column.

You can also use the tilde operator `~` for not, *flipping* the bits (from 1 to 0 and vice versa):

```
~0001 == 1110
```

```
~0011 == 1100
```

The result gives a one in a position if the input has a zero in that position, and a zero otherwise.

Figure 8-2 shows how ones and zeros combine for the and and or bitwise operations. The `|` percolates a one through, whereas the `&` percolates a zero through.

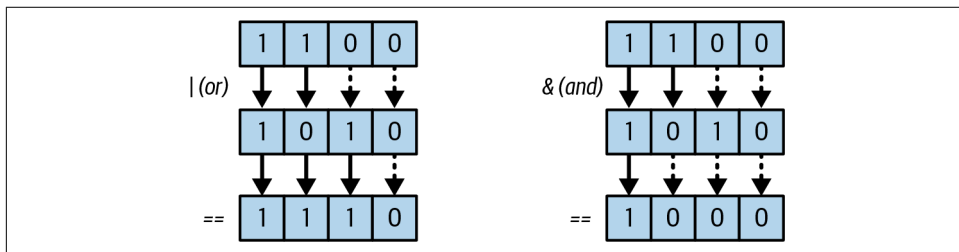


Figure 8-2. Bitwise operations

The filemodes use the bitwise or operator called a *bitmask* to detect if a mode is on or off. A bitmask uses the bitwise or to detect if specific bits are *ones*.

If the out mode is a 0001, and append, app is 0010, combining them gives:

```
0001 | 0010 == 0011
```

C++ can detect the out mode from 0011 using a bitwise & with mode 0001:

```
0011 & 0001 = 0001
```

The result is nonzero, so the out mode is on.

Similarly, you can detect the app mode using a bitwise & with mode 0010:

```
0011 & 0010 = 0010
```

The result is also nonzero, so the app mode is on.

There are other filemodes, including binary. By default, you get text mode, which means that if you open the file in an editor, you can read the contents. You can also write (and read) your file in binary mode, using `std::ios::binary`. The way the file is written is then implementation-defined, but in theory, you can make a file smaller this way. (The details are beyond the scope of this introductory book.)

Reading Previous Prices

At the moment, `read_from_file` displays prices to the screen. Let's write a more general version that returns the values in a `std::vector`. Put this new function in *input.cpp* (and remember to declare it in *input.h*). You need to include `<fstream>` and `<stdexcept>` in the source file, but you don't need other includes in the header. Add the declaration inside the namespace in the header:

```
std::vector<double> read_from_file(const std::string & filename);
```

Put the definition inside the namespace in the source file, as shown in [Example 8-3](#).

Example 8-3. Reading prices into a `std::vector`

```
std::vector<double> read_from_file(const std::string & filename) ❶
{
    std::ifstream file{filename};
    if(file)
    {
        return get_prices(file, [](){}); ❷
    }
    else
    {
        throw std::runtime_error("Failed to read to file");
    }
}
```

❶ Returns a `std::vector`

❷ Gets the values via your existing `get_prices` function

You will now be able to reuse this function in future chapters.

You can also avoid that jump in prices when you append to the file by finding the previous price in an existing *prices.txt* file, if there is one. Let's talk through the steps and then write the new code:

1. See if there's a *prices.txt* file already.
2. If this exists, read it and find the last price.
3. Simulate new prices, using the last price or, if there is no last price, \$100.00.
4. Append these to the existing *prices.txt* file or make a new file.

The filesystem library provides an `exists` function, which you can use to check if a file exists. Rearrange your `main` function and read an existing file, if there is one:

```
int main()
{
    try
    {
        const std::string filename{"prices.txt"}; ❶
        if(std::filesystem::exists(filename)) ❶
        {
            auto prices = stock_prices::read_from_file(filename); ❷
        }
        return 0;
    }
    catch(const std::exception & e)
    {
        std::cout << e.what() << '\n';
        return 1;
    }
}
```

❶ Checks if the *prices.txt* file exists

❷ Reads the prices if the file exists

There is a brief window of opportunity for someone else or another process to delete the file before you then try to read it, so this isn't fool-proof. Instead of writing and then reading prices, you're now reading prices first. You'll write the new prices shortly.

First, notice that you've left the `try/catch` block in place. The `read_from_file` function does throw an exception if there is a problem opening the file, so why do you need to check for the file as well? Because an exception could be thrown if the file does not exist or if another process has it open for exclusive access.

The file not existing isn't exceptional, so it's better to check for this potential problem first. Using exceptions to control the flow through a program will make your code hard to follow and reason about.

If you find a *prices.txt* file, you use the last price to simulate more. If there are no prices yet, just use a starting price of \$100.00. That's much simpler than trying to deal with an exception. If someone or something else is using the file, the problem is beyond your control, so recovering is difficult. That *is* exceptional, so stopping the program after reporting the problem is sensible.

Let's write some more prices out. Start with a default first price of \$100.00 and then try to read a prices file. Armed with the previous prices, you can take the last value, using back, provided there are some values:

```

double first_price{100.0}; ❶
const std::string filename{"prices.txt"};
if(std::filesystem::exists(filename))
{
    auto prices = read_from_file(filename); ❷
    if(!prices.empty()) ❸
    {
        first_price = prices.back(); ❹
        std::cout << "Read " << prices.size() << " prices\n";
    }
}

```

- ❶ Starts with a default price of \$100.00
- ❷ Reads the prices.txt file, if it exists
- ❸ Checks that prices were read from the file
- ❹ Uses the previous price, if there is one

You can now write new prices, appending them to any existing ones. Rather than using \$100.00, you'll use the `first_price`:

```
stock_prices::get_prices(first_price, 10, 0.05);
```

Let's look at all the changes. Here's the full listing:

```

#include <filesystem>
#include <fstream>
#include <iostream>
#include <vector>

#include "input.h"

void write_to_file(const std::vector<double> & prices,
                  const std::string & filename)
{
    const std::filesystem::path path = std::filesystem::current_path();
    const auto fully_pathed_filename = path / filename;
    std::ofstream file{filename, std::ios::out | std::ios::app}; ❶
    if(file)
    {
        for(auto price: prices)
        {
            file << price << '\n';
        }
        std::cout << "Wrote to " << fully_pathed_filename << '\n';
    }
    else
    {
        auto error_message = "Failed to write to " +
                              fully_pathed_filename.string();
    }
}

```

```

        throw std::runtime_error(error_message);
    }
}

int main()
{
    try
    {
        double first_price{100.00};
        const std::string filename = "prices.txt"; ❷
        if(std::filesystem::exists(filename)) ❸
        {
            auto prices = stock_prices::read_from_file(filename); ❹
            if(!prices.empty()) ❹
            {
                first_price = prices.back(); ❹
                std::cout << "Read " << prices.size() << " prices\n";
            }
        }
        write_to_file(stock_prices::get_prices(first_price, 10, 0.05), ❺
            filename); ❻
        return 0;
    }
    catch(const std::exception & e)
    {
        std::cout << e.what() << '\n';
        return 1;
    }
}

```

- ❶ Appends to a file (or creates a new one if it doesn't exist)
- ❷ Declares the filename in one place
- ❸ Checks if the file exists
- ❹ Reads from the file, using the last price, if any
- ❺ Writes further prices using `first_price` to start, which is either \$100.00 or the previous price in the input file
- ❻ Sends the filename to the `write_to_file` function

If you build and run your code now, you will see how many prices were read and where new prices were written:

```

Read 10 prices
Wrote to "/mnt/d/OReilly/introducing-c-plus-plus/code/chapter_08/prices.txt"

```

Try running the program a few times to see the file grow.

You can now generate lots of prices and save them directly to a file. You could even get some actual stock prices and save them to a file to read them back in for analysis.

You have learned even more C++, and now you've come to a point where you can reuse code you wrote before in different ways. Congratulations.

Conclusion

Files are streams, so you can use the stream insertion operator `<<` and stream extraction operator `>>` to write and read from files, as you've done with `std::cout` and `std::cin`. A file will try to open automatically for you and will close when the object goes out of scope. You can check a file has opened by using `if`.

You tried some parts of the filesystem library in this chapter: you got a `std::filesystem::path` from `std::filesystem::current_path()`, and you used operator `/` to join paths.

The `std::filesystem::path` has a `string` function, which gives you the path as a string. The filesystem has other methods too, such as `exists`, which you used.

You learned about file modes, which control how a file is opened, including:

- `std::ios::out` opens for output
- `std::ios::app` opens for appending

You can also use `std::ios::in` for input. By default, files are in text mode, but you can use `std::ios::binary` instead, though the details are beyond the scope of this book.

You join modes using the operator `|`. This uses the bitwise operator `or`. I showed you the other bitwise operators, `&`, `|`, and `~`, which operate on individual bits in a number. These are similar to the logical operators, `&&`, `||`, and `!`. You can read them as *and*, *or*, and *not*.

You also had extra practice with exceptions, throwing a `runtime_error` when you couldn't open a file. You added a `try/catch` block around the code in `main` to stop problems from leaking out of your code. You also saw how to explicitly return a value from `main`, using `0` (or `EXIT_SUCCESS`) to indicate no problems and any other number to indicate a problem.

You've covered a lot of ground so far. **Chapter 9** will teach you more details about strings and show you how to format your output nicely.

Strings and Formatting

You have used various types from the C++ standard library now, and you've learned about fundamental types like `int` and `double`. You have used characters, like `'\n'`, and messages, like `"Hello, world!"`, and even the `std::string` several times. I haven't explained the differences between these types in detail yet, however. This chapter will go through the different string types and how to use them correctly.

You'll learn about creating, manipulating, and formatting strings. I'll show you how to make a `string_view`, a view that does not change a string. You will also see how to pass command-line arguments (which are C-style strings) to `main` so you can vary a program's behavior.

You already have code to generate prices randomly and to read prices from a stream. In the hands-on part of the chapter, you'll use these functions to make an improved version of your trading game, **"Building a Trading Game" on page 143**, against the various prices. This version will allow you to buy or sell stock, as well as displaying the prices in a neater way: I'll show you how to print the prices to two decimal places, displaying them as `"$100.00"` rather than `"100."`

C-Style String Literals and Characters

Make a new file called `main_with_args.cpp`. You'll use it to experiment with the code in this section. I will provide some information on various string types first and then show you how to use them in a `main` function in the next section.

A single character uses single quotes (`'`) and has type `char`, which is a numeric type used to represent a character. It sounds weird, but computers deal with numbers, so ultimately everything must be a number. For example, an `'A'` corresponds to the ASCII value 65. You can use either to declare the letter A:

```
const char letter_from_character{'A'};
const char letter_from_number{65};
```

Using the 'A' version is clearer, but in both cases a 65 is used in memory.

Characters in double quotes are C-style *string literals*. A literal is *literally* embedded in the program. If you look at the contents of the build of [Example 1-3](#), one of your first programs, you will see `Hello, world!` in the binary file.



The C++ language derives from the C language. C++ originated as “C with classes”, making it a superset of C. You can still use C in C++ code, but the languages have diverged in a few places.

A string literal is composed of individual characters. You can access these characters using the operator `[]`, just as you do for individual elements in a `std::vector`.

Now, the string `Hello, world!` contains 13 characters. The C programming language uses a null character, `\0` (the ASCII value 0), to indicate the end of a string. Thus, the message is composed of 14 individual char values, as shown in [Figure 9-1](#).

String literal	'H'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'
Char index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 9-1. Individual characters that comprise a string literal include a null at the end

You can declare the message to be a C-style *array* using `[]`.

```
const char message[]{"Hello, world!"};
```

Optionally, you can provide the array’s size in the `[]`, but you need to leave space for the terminating null, making it 14 characters. That’s easy to forget, so it’s safer to avoid specifying the size. Unlike `std::array`, the C-style array doesn’t have a size method. This makes C-style arrays (including character arrays) harder to work with.

You can also declare a C-style string literal as a *char pointer*:

```
const char *message{"Hello, world!"};
```

A pointer is somewhat like an iterator: it indicates a position. The string literal is a `const char[14]`, but the char pointer is just pointing to a character. This is described as a C-style array *decaying* to a pointer. Though this sounds somewhat rotten, it means that the information about the array size is lost. The C-style array is better because it retains this information, but you can’t always choose. In general, the C++ `std::string` is even better, because with it, you don’t need to know all the C features.

You can even create a C-style array of C-style string literals:

```
const char * messages[] {
    "Hello, world!",
    "How are you?"
};
for(auto message: messages)
{
    std::cout << message << '\n';
}
```

If you try this, you will see both messages print. Avoid using it in your code, though: `std::vector` and `std::string` are easier to work with. I include it here because a C-style array of char pointers is used to pass arguments to `main`.

Providing Arguments to main

Command-line arguments are arguments from the command line (or prompt). You can use them to pass values to `main`. To do so, you'll use a different signature for `main` that takes two parameters: a count (an `int`) and arguments (values sent to a program, which here are an array of char pointers):

```
#include <iostream>

int main(int argc, char *argv[])
{
    for(int i = 0; i<argc; ++i)
    {
        std::cout << argv[i] << '\n';
    }
}
```

Try this code in your *main_with_args.cpp* file.

The variable names you use for the count and arguments don't matter, but you often see `argc` for the argument count and `argv` for the argument values. Some people use a pointer to the char pointers used instead of an array of char pointers:

```
char ** argv;
```

There is always at least one argument: the program's name, which appears at position 0. Subsequent arguments are at positions 1, 2, ... `argc-1`.

When you run this program, you can provide some arguments after the program name, which will be displayed:

```
./main_with_args first second third
./main_with_args ❶
first ❷
second
third
```

- ❶ Shows the program's name
- ❷ Shows subsequent command-line arguments

In [Example 8-1](#), you used the hardcoded string literal `"prices.txt"` for a filename. You can now change your code to accept the filename as a command-line argument. This lets you use the same program with various files, so you won't need to recompile it if you want to use a different file.

Working with C-style arrays and string literals is difficult. You need to pay lots of attention to indexing, ensuring any index is valid. For C-style strings, you need to deal with low-level memory management. Fortunately, the C++ `std::string` deals with resizing for you. Avoid declaring C-style arrays of char pointers where possible, because they require such careful handling. However, they are needed sometimes—for example, when using command-line parameters. Let's look at the C++ `std::string`, in contrast to string literals.

Creating and Manipulating a `std::string`

You have used a `std::string` a few times. You can declare a `std::string` and provide a literal for initialization:

```
const std::string greeting{"Hello, world!"};
```

If you used `auto` instead, the greeting type would be a char array:

```
const auto greeting{"Hello, world!"};
```

That could prove annoying or lead to buggy code. For example, since you can overload functions in C++, you might end up calling the wrong function. So the type matters. In [“Almost always auto” on page 51](#), I noted many people say almost always `auto`. There are some cases where `auto` can deduce the wrong type.

You can state that you want a `std::string` by adding a *suffix* of `s` to the literal. The suffix `s` is actually the operator `""s` defined in the string header, so you need to make it clear that you mean that operator:

```
using namespace std::string_literals; ❶  
const auto yet_another_greeting{"Hello, world!"s}; ❷
```

- ❶ Says to use the `string_literals` namespace, to find operator `""s`
- ❷ Creates a `std::string`, rather than a C-style string literal



Stating that you want to use a namespace here makes the calling code neater. You can immediately see "Hello, world!"s without namespaces and scope-resolution operators in abundance.

You've met other suffixes for numbers, like `0u` for unsigned, in “[Understanding Code with Random Numbers \(and Vectors\) in Depth](#)” on page 146.

Other Ways to Create a `std::string`

Some `std::string` creation methods are similar to a `std::vector`.

Let's consider a few examples:

```
std::string from_a_literal{"abc"}; ❶  
std::string from_some_chars{'a', 'b', 'c'}; ❷  
std::string triple_A(3, 'A'); ❸  
std::string triple_A_from_char_value(3, 65); ❹
```

- ❶ Creates a string from a literal
- ❷ Creates a string from individual chars
- ❸ Creates a string with three As
- ❹ Also creates a string with three characters using the ASCII for A (65)

Apart from the creation from a literal, the initializer list, count of a value, and count of an ASCII value are all ways to create a `std::vector`:

```
std::vector from_some_chars{'a', 'b', 'c'}; ❶  
std::vector triple_a(3, 'A'); ❷  
std::vector<char> triple_a_v2(3, 65); ❸
```

- ❶ Creates a vector from individual chars
- ❷ Creates a vector with three As
- ❸ Also creates a vector with three characters ASCII for A (65), but notice you have to specify `char`

You can think of a `std::string` as being very like a `std::vector<char>`. You can `push_back` characters, ask for the first character with `front`, and use `[]` to find the character at a specific index number.



The `std::string` has a lot of member functions and can be created in various ways. It's easy to get lost in all the possible ways to create and manipulate strings. If you know how to create a `std::string` and you remember that `[]` lets you access individual characters, you have enough to write useful C++.

More `std::string` Functions

Let's try out some `std::string` functions.

Add a new function to *main_with_args.cpp*, above `main` and call it `show_characters_before`. It will take a `std::string` by `const` reference. As you know, this avoids making a copy. It will also take a `char`. You can then search for a chosen character (for example, a slash in a file path) and display everything before that character in the `std::string`.



If you want to work with file paths and directories, don't forget about the standard filesystem library. This section shows the extra work you need to do without that library, as well as introducing some generally useful `std::string` functions.

You can use the `std::string` method `find` to find a character. Unlike the standard algorithms, this returns a numeric value, rather than an iterator. If the character is not present, a special value, `std::string::npos` (for “no position”), is returned. You can then make a substring using the `substr` member function to form a new `std::string` containing the characters before it, like this:

```
#include <iostream>
#include <string> ❶

void show_characters_before(const std::string & value, ❷
                           char character) ❸
{
    auto position = value.find(character); ❹
    if(position != std::string::npos) ❺
    {
        std::string partial = value.substr(0, position); ❻
        std::cout << "Before " << character << ": " << partial << '\n'; ❼
    }
    else
    {
        std::cout << character << " not found\n";
    }
}

int main(int argc, char *argv[])
{
```

```

    for(int i = 0; i<argc; ++i)
    {
        std::cout << "Argument " << argv[i] << '\n';
        show_characters_before(argv[i], '/'); ❸
        show_characters_before(argv[i], '\\'); ❹
    }
}

```

- ❶ Includes the string header
- ❷ Defines a function, taking a string by const reference
- ❸ ... and a character
- ❹ Tries to find the required character
- ❺ Compares with npos
- ❻ Creates a new string from a substring starting at 0, up to, but not including the position character
- ❼ Displays the substring
- ❽ Passes the i^{th} argument as a string, trying to find a forward slash
- ❾ Passes the i^{th} argument as a string, trying to find a backslash

Build and run your code and try a few command-line arguments. If I use help and apple/banana/cherry as my arguments, I get the following:

```

$ ./main_with_args.exe help apple/banana/cherry
./main_with_args.exe
Before /: .
\ not found
Argument help
/ not found
\ not found
Argument apple/banana/cherry
Before /: apple
\ not found

```

Don't forget: the first argument is always the program name.

There are four things to notice about this code:

- First, the function takes a `std::string`, but you pass it a `char *`. A `std::string` can be created from a `char *`, so this is OK.

- Next, the `char` isn't passed as a `const` reference. You have passed `int` and other built-in types by value in a similar way, but you used `const` references for larger objects, like `std::vector`. This is a common pattern. Fundamental types—that is, the simple types built into the language—should be passed by value, whereas large objects should be passed by reference to `const`.

You *could* use `const` references for everything, but each reference takes some space, maybe more than a `char`. This won't make much difference to your program, but it can matter in embedded systems with less memory:

- The `substr` function uses a half-open range: `[begin, end)`. You have seen this used for other container types.
- Finally, to declare a forward slash, you can use `'/'`. A backslash, however, needs another backslash to escape the character `'\\'`, just as you use `'\n'` to indicate that the `n` is a special character.

Now, you might not want a whole new string to use a substring. You would then have *two* strings, and the original already contains the characters you want. C++17 introduced a `std::string_view`, which allows you to use a part of an existing string—which could be a string literal, a `std::string`, or even a `std::vector<char>`. Let's have a look.

String Views

You met some views when you used the ranges library. These are *lazy* views, as you saw in “[Lazy Views](#)” on page 132: they are evaluated only when used. Furthermore, views don't copy the data. The `std::string_view`, from the header `<string_view>`, gives you a view of a string but also has the advantages of lazy loading and not copying the data. In fact, it can view more than just a `std::string`—you can have a view of any contiguous sequence of characters.

Let's rewrite the `show_characters_before` function to use a `std::string_view`, instead of making a new string from a substring.

The new version of `show_characters_before` can take a `std::string_view` by copy, because the view refers to an existing contiguous sequence of chars already. You need to include `<string_view>` to use the `std::string_view`. Apart from that, the new version looks very similar to the old version:

```
#include <iostream>
#include <string_view> ❶

void show_characters_before(std::string_view value, ❷
                           char character)
{
```

```

    auto position = value.find(character);
    if(position != std::string::npos)
    {
        std::string_view partial = value.substr(0, position);
        std::cout << "Before " << character << ": " << partial << '\n';
    }
    else
    {
        std::cout << character << " not found\n";
    }
}

int main(int argc, char *argv[])
{
    for(int i = 0; i<argc; ++i)
    {
        std::cout << "Argument " << argv[i] << '\n';
        show_characters_before(argv[i], '/'); ❸
        show_characters_before(argv[i], '\\'); ❹
    }
}

```

- ❶ Includes the `string_view` header
- ❷ Defines a function, taking a `string_view`
- ❸ Passes the i^{th} argument as a `string_view`, trying to find a forward slash
- ❹ Passes the i^{th} argument as a `string_view`, trying to find a backslash

If you build and run the code, you won't notice any different behavior. The important difference is that, previously, `partial` made a new string. Now it uses a view of the existing value instead.

Don't forget, the `std::string_view` is a view. So what happens if you change what's being viewed? Well, if you take a view of a string and change the original string, the view is invalidated. For example, consider this code:

```

using namespace std::string_literals;
std::string some_string("Original string"s);
std::string_view view_of_some_string{some_string}; ❶
std::cout << view_of_some_string << '\n'; ❷
some_string = "A different string"; ❸
std::cout << view_of_some_string << '\n'; ❹

```

- ❶ Takes a view of a string
- ❷ Displays the view

- ❸ Changes the original string
- ❹ Tries to display an invalid view

Displaying the view when the `std::string` is unchanged is fine.

But change the `std::string` and then try to display the view again. That view is no longer valid, so you're likely to see nonsense: a fragment of text or something like that. Here's what I get:

```
Original string
inal string
```

Figure 9-2 hints at what has happened: when the string being viewed changes, the view is looking at memory that no longer holds the previous characters. Viewing something that no longer exists is undefined behavior.

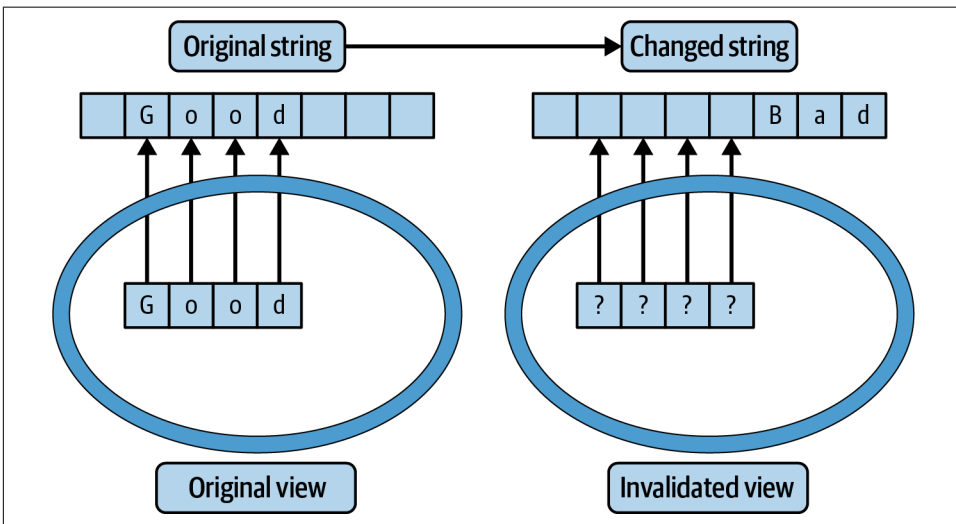


Figure 9-2. Changing what a `string_view` is viewing invalidates the view

In Figure 7-4, you saw how a `std::vector` can reallocate as you add elements. The `std::string` behaves in a similar way, invalidating iterators and views. So be careful when you use a `std::string_view`. Make sure whatever it is you're viewing stays in scope and doesn't change.

Now you can pass arguments to `main`, and you've practiced using strings and string views. Let's work to improve the trading game from Example 7-2 by formatting the prices properly.

Formatting and More on std::println

Create a new *main.cpp* file. Start by generating some prices, using your *input.h* file. Add the *input.cpp* file to your build:

```
#include "input.h"

int main()
{
    const auto prices = stock_prices::get_prices(100.0, 10, 0.05);
}
```

Previously, you used `std::cout` to display prices. Back in [Chapter 1](#), you used `std::println`, but you haven't used it since. It's much easier to use `std::print` and `std::println` to format output than to use `std::cout`, though, so let's do just that to display prices.

Using the `fmt` library

Some older compilers do not support this way to format yet. If you can't get `std::format` or `std::println` to work locally, you can use the `fmt` instead. You need to install the library and include `<fmt/base.h>` to use it in your code. This library provides `format`, `print`, and `println` in the namespace `fmt`, so you need to use `fmt::` instead of `std::` in the code listings in this chapter if you need to use `fmt`. The docs provide a link to [Godbolt](#) if you want to try it online.

You used `std::println` with a string literal:

```
std::println("Hello, world!");
```

To use a variable, you can put `{}` in the double quotes, called a *replacement field*, and give the variable to place in the field after the string. For your prices, you can do this in a range-based for loop:

```
for(auto price: prices)
{
    std::println!("{}", price);
}
```

This will have the same effect as using `std::cout` and adding the newline character yourself:

```
for(auto price: prices)
{
    std::cout << price << '\n';
}
```

You will see some prices, displayed to several decimal places:

```
97.09842865845474
100.59736283913799
90.8913818169921
88.57896314980702
90.70721381119712
86.80768016551397
93.1898521426114
92.01385524250107
96.78098098429128
94.64386319404912
```

You can add formatting requirements inside the `{}`. Use `:.2f` to ask for two figures after the decimal point for floating-point numbers:

```
for(auto price: prices)
{
    std::println("{:.2f}", price);
}
```

Now your prices show two decimal places:

```
95.31
108.50
99.73
90.89
92.29
88.92
82.48
86.29
90.06
86.86
```

Let's look at the `{:.2f}` magic in more detail.

std::format and Format Specifications

The special string you used for `std::println` is called a *format string*: a string with instructions for formatting. `std::println` prints to a stream. You can use `std::format` from the `<format>` header to create a formatted string instead.

Here's a small example:

```
std::string display_price = std::format("{:.2f}", price);
```

I'll talk you through more details and then show you how to use these in an extended trading-game function.

If you've used Python before, formatting strings using `{}` might be familiar. You provide a string, which can be a message, a format string, or a mix of both. This string gives you ways to format or alter the appearance of the output. The format string can

contain `{}` to indicate a replacement field. You add variables, separated by commas, after the format string. Inside the `{}`, you can refer to the index of specific variables before a `:` and add formatting after the `:`.

Indexing starts from 0, just as it does to access an element in a vector or a position in a string. Without an index, the variables are used in order. Here's an example:

```
std::println("{} {}", 1, 2); ❶
std::println("{0:} {1:}", 1, 2); ❷
std::println("{1:} {0:}", 1, 2); ❸
std::println("{1:} {0:} {1:}", 1, 2); ❹
```

- ❶ Uses the variables in order in the replacement fields
- ❷ Uses variable with index 0 (the first number) and then index 1
- ❸ Uses variable with index 1 (the second number) and then index 0
- ❹ Uses the second variable twice

The output is as follows:

```
1 2
1 2
2 1
2 1 2
```

There are various formatting options. You used `.2f` to show a floating-point number to two decimal places. [CppReference](#) gives details of what else is possible. For example, you can fill the string with spaces, or any other character, or align it to the left, right, or center (I'll show you how in the next section).

Importantly, many mistakes give an error at compile time. That's much better than getting scrambled output or undefined behavior. You can provide too many arguments, but if you don't provide enough, you get an error.

Try an example, like this:

```
auto too_many = std::format("{0:} {1:}", 1, 2, 3);
auto too_few = std::format("{0:} {1:}", 1);
```

The first line will compile, but the second gives an error:

```
error: call to non-'constexpr' function
      'void std::__format::__invalid_arg_id_in_format_string()'

```

The message is slightly cryptic, but it does say “invalid arg id in format string,” which is the case. `too_few` uses an index of 1, but there is only one argument, so the largest index is 0.

You can add more than just the format specifier in the quotes. Anything outside the {} will be used verbatim. So you could use a dollar sign for your prices:

```
std::string display_price = std::format("${:.2f}", price);
```

You can add more than a single dollar character, though; you can add whole messages on either side of the replacement field.

Let's use `std::format` and `std::println` to make a tidier trading game, learn more about formatting, and pull together what you've learned so far in this chapter.

An Improved Trading Game

Make a new *main.cpp* file. You are going to write another trading game. It will be based on “[Building a Trading Game](#)” on page 143, but this time, you can buy or sell stock.

Your program will use command-line arguments to (optionally) provide a filename from which to load prices. If you don't provide a filename, the program will generate random prices. You therefore need to add *input.cpp* in your build so you can use the functions you wrote to load or generate prices.

The game will start with initial funds of \$100.00 and no shares. It will display each price and the current status, showing the player's funds and number of shares. The player (you, or anyone else playing) can then buy or sell, but your program needs to check that they have enough funds (or stock, as the case may be). Alternatively, the player can stick as they are. The game continues with a price update, followed by the chance to buy, sell, or stick. At the end of the prices, it will show the total profit, and then the game is over.

You wrote a `read_from_file` function in [Example 8-3](#). You can call this from `main`, if you pass a filename at the command line. Otherwise, you call `get_prices` to generate random prices. Rather than using `if/else`, you can use the ternary operator, `condition ? true_value : false_value`. Both of these functions are in the `stock_prices` namespace. You could add `stock_prices::` before each call, but you can also just say you're using this namespace, as the following code shows:

```
#include "input.h"

int main(int argc, char *argv[])
{
    using namespace stock_prices; ❶
    const auto prices = (argc>1) ? read_from_file(argv[1]) : ❷
                                get_prices(100.0, 10, 0.05); ❸
    trading_game(prices);
}
```

- ❶ States that you want to use the `stock_prices` namespace

- ❷ Uses the second command-line argument, if there is more than one argument
- ❸ Generates random prices otherwise



You often see code with `using namespace`, since this saves you from having to spell it out each time. It brings names into the current scope. However, this could mean that two functions from different namespaces are visible, causing an ambiguity. For this reason, avoid putting a `using` statement somewhere with a wide scope, like a header file. Inside a function is fine, but you may still need to use a namespace and the scope-resolution operator to disambiguate identical names.

Now you can create your new trading game. Add a new function for this, above the `main` function, in *main.cpp*. You need the prices obtained in `main`, so pass these by `const` reference: they will only be read, so they can be constant, and using the reference avoids copying them. The player will start with an initial pot of cash, and you'll need variables to track the funds available and the number of shares they buy.

Your program will display the prices one at a time. The player can type `s` to sell, `b` to buy, or any other character to continue. When the prices run out, you can find the difference between the current funds and the `initial_funds` to show the profit (or loss).

Add the implementation of the game to the function, as shown in [Example 9-1](#).

Example 9-1. The new trading game

```
#include <iostream>
#include <print>
#include <stdexcept>

#include "input.h"

void trading_game(const std::vector<double> & prices)
{
    const double initial_funds{100.0};
    double funds{initial_funds};
    int number_of_shares{};

    for(auto price : prices)
    {
        auto status = std::format("Funds {:.2f}, Shares {}",
                                  funds, number_of_shares); ❶
        std::println!("{}", status);
        auto price_message = std::format("Current price: {:.2f}", price); ❷
        std::println("{}: >{}", price_message, status.size()); ❸
    }
}
```



```

std::println("Press (s) to sell, (b) to buy");
std::print("or something else to continue>"); ❹
char choice{};
std::cin >> choice; ❺

if (choice == 's') ❻
{
    if(number_of_shares > 0) ❼
    {
        --number_of_shares;
        funds += price;
    }
    else
    {
        std::println("No stock to sell");
    }
}
else if(choice == 'b') ❸
{
    if(price <= funds) ❾
    {
        ++number_of_shares;
        funds -= price;
    }
    else
    {
        std::println("Insufficient funds");
    }
}
}
std::println("Total profit ${:.2f}", funds - initial_funds); ❿

```

- ❶ Makes a string first, so you can find its length
- ❷ Formats the price to two decimal places
- ❸ Pads the next message to match the length of the previous one and right-justifies it
- ❹ Prints the message without a new line
- ❺ Gets the player's choice
- ❻ Sells shares if s is pressed
- ❼ Checks that there are some shares to sell and either updates the funds and number_of_shares or reports a problem
- ❾
- ❿

- 8 Buys shares if b is pressed
- 9 Checks that there are sufficient funds to buy shares and either updates the funds and number_of_shares or reports a problem
- 10 Displays the profit (or loss)

There are various ways to use format strings to make the display neat. Before you call the new function, let's look at the formatting in [Example 9-1](#).

You use `{:.2f}` to display the price to two decimal places in the status:

```
auto status = std::format("Funds ${:.2f}, Shares {}",
                          funds, number_of_shares);
std::println!("{}", status);
```

The `number_of_shares` has no special formatting. You then display the status. Since you're using a `std::string`, you can find out how long the message is using `status.size()`.

This means you can align the next message with the status message:

```
auto price_message = std::format("Current price: ${:.2f}", price);
std::println("{: >{}}", price_message, status.size());
```

After the `:`, you have a space and a `>`, and a second pair of `{}` inside the outer `{}`. This second `{}` is called a *nested replacement field*, because it is nested inside an outer `{}`. You can then provide a value, for example, the `status.size()`, as shown previously. The space and chevron mean *justify*, which pads the text with spaces to make it even on both sides. To left-justify, you can use `<`, and you can center the text using `^`. You can use characters other than spaces if you want.

To pad a message, you need to say what field length you require. You can give a fixed size: `{:*>6}` would pad a field with asterisks at the start to make it six characters, with the value at the right. Alternatively, you can use a nested replacement field with a `{}` instead of the 6, which means you can vary the field width to something other than 6. You used the `status.size()`, so the `price_message` and `status` messages are right-aligned.

The final part of the output prints a message and then uses `std::print` to avoid the newline character, leaving a `>` as a prompt:

```
std::println("Press (s) to sell, (b) to buy");
std::print("or something else to continue>");
```

Now you need to call the game from `main`:

```
int main(int argc, char *argv[])
{
    using namespace stock_prices;
    const auto prices = (argc>1) ? read_from_file(argv[1]) :
                                get_prices(100.0, 10, 0.05);

    trading_game(prices); ❶
}
```

❶ Calls the new game

Build your code and you can play your game. You should have a *prices.txt* file from [Chapter 8](#), which you can use as an argument to your program. I called mine *trading_game.exe*:

```
$. /trading_game.exe chapter_08/prices.txt
```

Alternatively, you can run the program without command-line parameters. If you do, the game uses randomly generated prices.

Here's a sample of the end of the game, after I bought shares for \$87.60 and sold them for \$89.01:

```
Funds $101.40, Shares 0 ❶
  Current price: $79.48 ❶
Press (s) to sell, (b) to buy
or something else to continue>c ❷
Total profit $1.40 ❸
Game over
```

❶ The two lines are right-aligned, with spaces at the start of the current price.

❷ I pressed c to avoid buying or selling.

❸ The total profit is displayed, to two decimal places.

Now, the profit should be \$1.41 (\$89.01 – \$87.60), not \$1.40. This is because the values are shown to two decimal places. The values in the *prices.txt* file are \$89.0052 and \$87.6022, which gives \$1.403, so this value is displayed as \$1.40. The format specifier rounds decimals, so you should really save the prices to two decimal places either before you use them in the game or before you calculate the profit.

Next, you will use `std::println` to save the prices to two decimal places, so the right profit will be reported.

Understanding Strings in Depth

A `std::string` is an instance of the more general class template `std::basic_string`, using `char` as the type: `std::basic_string<char>`. This is suitable for ASCII characters and some UTF8 characters, including some emojis and accents. C++ also provides a `wchar_t` type, for *wide* characters. Wide characters take up more space but allow you to use a wider variety of characters. The corresponding wide string is a `std::wstring`, which means `std::basic_string<wchar_t>`.

Dealing with non-ASCII characters in a way that works on all platforms is difficult. C++ does support UTF8, but you might have to change your console's locale. When you use `std::cin` and `std::cout` on Windows, you need to set the codepage, using `SetConsoleCP` for input and `SetConsoleOutputCP`, and use the flag `/utf-8` when compiling. In contrast, `std::print` supports UTF8 directly. C++23 has added more support for Unicode (for example, see [Sandor Dargo's blog](#)). However, you may still get varying behavior between operating systems.

Joining `std::strings` Efficiently

You can use the addition operator to concatenate `std::strings`:

```
using namespace std::string_literals;
auto message = "Hello,"s + " world!"s;
```

The `std::string` `message` then contains `"Hello, world!"`.

If you add more strings, you'll end up with unnamed temporary objects. A *temporary object* is an object that is created while evaluating an expression. It is created during the expression's evaluation and then goes out of scope. This isn't a disaster, but it isn't very efficient, either.

Let's consider what happens when you add several strings:

```
using namespace std::string_literals;
auto message = "Hello"s + " again,"s + " world!"s;
```

First, `"Hello"` and `" again,"` are joined to form the temporary string `"Hello again,"`. This temporary string is then joined with the `" world!"` string to give the result. The `message` uses five strings: the final string, the original three, and a temporary. In effect, you are doing this:

```
using namespace std::string_literals;
auto first_temporary = "Hello"s;
auto second_temporary = " again,"s
auto third_temporary = " world!"s;
auto fourth_temporary = first_temporary + second_temporary;
auto message = fourth_temporary + third_temporary;
```

Using `std::format` is more efficient:

```
using namespace std::string_literals;
auto message = std::format("{}{}{}", "Hello"s, " again"s, " world!"s);
```

Now you have only four strings, because `format` builds up the output string for you.

Using `std::println` to Save to a File

So far in this chapter, you have used `std::println` and `std::print` to write output to the screen. C++ provides overloads of these functions that print to an output file stream. This means you can save your prices to the required number of decimal places, which will avoid the problem you saw in [Example 9-1](#). Try the improved function shown in [Example 9-2](#) in *main.cpp*.

Example 9-2. Using `std::println` with files

```
#include <filesystem>
#include <fstream>
#include <stdexcept>

void write_to_file_improved(const std::vector<double> & prices,
                           const std::string & filename)
{
    const std::filesystem::path path = std::filesystem::current_path();
    const auto fully_pathed_filename = path / filename;
    std::ofstream file{filename};
    if(file)
    {
        for(auto price: prices)
        {
            std::println(file, "{:.2f}", price); ❶
        }
        std::println("Wrote to {}", fully_pathed_filename.string()); ❷
    }
    else
    {
        throw std::runtime_error(
            std::format("Failed to write to {}", ❸
                        fully_pathed_filename.string() ❹
                        )
        ); ❺
    }
}
```

❶ Uses the format specification along with `println` to write to a file

❷ Uses `println` to indicate successful completion

- ❸ Uses `format` to give an error message
- ❹ Throws a runtime error using the message

To run it, you need to call it from `main`, after you obtain prices:

```
write_to_file_improved(prices, "prices.txt");
```

There is an item of note in [Example 9-2](#). You used `fully_pathed_file`.`name.string()` to get the path as a `std::string` for use in `std::format` and `std::println`. C++26 introduced a formatter for the filesystem path, meaning you won't need to call the `string()` function. Compilers are starting to offer C++26 features as I write this in mid-2025, but only a few are available at the moment. At some point, compilers will support this:

```
std::println("Using string {}", fully_pathed_filename);
```

You would need to build using `-std=C++26` instead of `-std=C++23`. Currently you can use the `fmt` library instead, like this:

```
#include <filesystem>
#include <fmt/std.h> ❶
#include <print>

int main()
{
    const std::filesystem::path path = std::filesystem::current_path();
    const auto fully_pathed_filename = path / "prices.txt";
    std::println("Using string {}", fully_pathed_filename.string()); ❷
    fmt::print("fmt path: {}\n", fully_pathed_filename); ❸
}
```

- ❶ Includes `fmt`'s `std.h`, for standard formatters
- ❷ Uses a formatter for a `std::string`
- ❸ Uses a formatter for a file path directly

You can try this on [Godbolt](#) if you haven't installed `fmt` or use the instructions in [“Using the `fmt` library” on page 187](#).

The prices are now saved in a file to two decimal places:

```
105.01
111.41
109.35
110.19
111.68
111.84
108.99
```

```
109.87
110.48
98.27
```

Now you will calculate the profit using the same value you display. This solves one problem. However, floating-point numbers have a maximum size and can support only a limited number of digits. Try displaying a number with lots of digits:

```
std::println!("{}", 1234567890123456.905);
std::println("{:.2f}", 1234567890123456.905);
```

The number is rounded after the 16th digit, so the output looks like this:

```
1234567890123457
1234567890123457.00
```

The 90 (and a half) cents have been rounded.

Floating-point numbers can be hard to handle, but if your prices aren't astronomical, you won't see these problems. A friend, Richard Harris, wrote an article called “Why Fixed Point Won't Cure Your Floating Point Blues” in *Overload*, 18(100):14-21, December 2010, which goes into some detail.

Conclusion

You've seen how to pass arguments to `main`, allowing you to, for example, provide a filename to read prices from rather than hardcoding the prices. The second version of `main` takes a count of arguments and a C-style array of char pointers:

```
int main(int argc, char *argv[])
{
}
```

The first argument is the program name, so `argc` is always at least one.

You learned about string literals and chars and also used the suffix `s` to make a `std::string`:

```
using namespace std::string_literals;
std::string message{"A proper std::string"s};
```

This introduced the idea of stating you want to use a specific namespace in a block. You did this when you used functions from the `stock_prices` namespace. You had a `using` statement first:

```
using namespace stock_prices;
const auto prices = (argc>1) ? read_from_file(argv[1]) :
    get_prices(100.0, 10, 0.05);
```

Without the using statement, you would have to specify the namespace:

```
const auto prices = (argc>1) ? stock_prices::read_from_file(argv[1]) :  
                             stock_prices::get_prices(100.0, 10, 0.05);
```

You've also seen several aspects of `std::string` in this chapter. This type is easier to use than C-style character arrays and behaves like a `std::vector`. There are a few differences; for example, the `find` function returns a number, using `std::string::npos` when the characters are not present. Though you can concatenate `std::strings` using operator `+`, using `std::format` is more efficient because it avoids making several temporary strings.

You also saw `std::string_view` and how it can leave you with an invalid view if you're not careful. It is useful nonetheless, because it can avoid copying characters and ending up with the same substring in two or more places.

You also used format strings with both `std::println` and `std::format`. To put a variable in a message, you use `{}`, giving a replacement field:

```
std::println("Message: {}", "some message");
```

You can format fields in various ways. You can add a number before a colon in a format specifier, stating which variable to use. This means you can use a variable more than once:

```
std::println("{1:} {0:} {1:}", 1, 2);
```

You then see `2 1 2`, because the value 2 is used first and last.

Formatting goes after a `:` in the `{}`. For example, to use padding and alignment, you give a character, like a space, the width required, and `>` to align right:

```
std::println("Message: {: >20}", "some message");
```

The output will make the "some message" field 20 characters long, padded with leading spaces:

```
Message:           some message
```

`<` aligns text to the left, and `^` centers it. You can use a nested replacement field rather than hard-coding 20. You nest more `{}` inside the replacement field, `{: >{}}`, giving the length as another argument to `std::println` or `std::format`.

You also used `{:.2f}` to show a number to two decimal places. I called out some issues that can happen when you round numbers or try to use very large values, but for reasonable prices, you won't get problems.

The format specifiers work for `std::format` as well, giving you a `std::string` rather than directly displaying a message. [CppReference](#) gives further details on format specifiers.

You also used `std::println` to write to a file, using the same format specifiers.

You practiced writing more functions, learning a common pattern:

- Fundamental types—that is, the simple types built into the language—should be passed by value.
- Large objects, like a `std::vector`, should be passed by reference to `const`, unless you need a copy.

You have now covered a lot of C++—but you haven’t written a class yet. In [Chapter 10](#), you will start to learn about classes in C++, so you will be able to generate prices for different stocks.

Classes: Member Variables and Member Functions

You now know about various fundamental types in C++, including numerical types, characters, and strings. You have used classes from the standard library a few times, including class templates like `std::vector`. This chapter will show you how to write your own classes.

Classes allow you to bundle together related elements. For example, you could bundle a stock name together with a starting price and a way to generate or input price updates. Namespaces also group related functions and types, but classes give you more options. Bundling elements into a class gives you a new type, which allows you to add new stock types to your trading game easily. You also give classes names, conveying meaning, which can make your code easier to read.

Around 1979, C++ started as “C with classes,” designed by Bjarne Stroustrup. He found classes useful for working on larger codebases because they provide a way to think at a higher level. You can think about a *stock*, rather than a name, a price, and a way to get a new price. Furthermore, the name is a `std::string` class, which is a higher-level concept than a C-style `char` pointer. As you have seen, there is much more to C++ than classes, but they are a fundamental part of many C++ codebases.

Let’s create a class first, starting with some data and then adding behavior. By the end of this chapter, you will be able to create a `std::vector` of various stocks. You will then be a step closer to a bigger trading game.

A Simple Class

A *class* is a user-defined type that can have member variables and member functions. The simplest form of a class is a *struct*.

Make a new file called *stock.h*. You are going to define a *Stock* class, with a name, last price, and volatility. You will extend this over the rest of the chapter and use it to generate prices for various stocks.

Declare a *struct* in your header, using the keyword *struct* followed by a name. For a simple *struct*, you can use curly braces, {}, to initialize the data members, which is called *brace initialization*. It gives each type a default value, for example, 0.0 for a *double*. The final curly brace for the class ends with a semicolon, like this:

```
#pragma once

#include <string>

namespace stock_prices
{
    struct Stock ❶
    { ❷
        std::string name{}; ❸
        double last_price{}; ❹
        double volatility{}; ❺
    }; ❻
}
```

- ❶ Declares a *struct* called *Stock*
- ❷ Starts the *Stock* definition
- ❸ Declares a *std::string* member called *name*
- ❹ Declares a *double* member called *last_price*
- ❺ Declares a *double* member called *volatility*
- ❻ Closes the definition, using a *}* and a semicolon

The *Stock struct* defines an *object type*. You can make specific instances with different member values. Similarly, an *int* is a specific type, and you can declare several *ints* with different values.

Start a new *main.cpp* file. Let's create a specific *Stock* using brace initialization, like this:

```
Stock coffee{"Coffee", 4.8, 11.3};
```

Using the initializer list to set the starting values is called *aggregate initialization*. The `Stock` aggregates together various members, and for simple structs, you can use the list to initialize the member values.

The `Stock` remembers the member values so you can retrieve them when needed. You use a dot (or period) to access the member variables in a class—you’ve seen that before. For example, you used a dot in [Chapter 2](#) to call the `eof` function member of `cin`: `std::cin.eof()`. You’ve also used a dot to access the member variables and functions of various standard library classes.

Now create an *instance* of a `Stock` (that is, a `Stock` with specific values) inside `main`, and use the dot operator to access the members. Include `<iostream>` to display the values:

```
#include <iostream>

#include "stock.h"

int main()
{
    stock_prices::Stock coffee{"Coffee", 4.8, 0.0113}; ❶

    std::cout << coffee.name ❷
               << ": price " << coffee.last_price ❷
               << '\n';
}
```

❶ Creates an instance of `Stock` with specific values

❷ Displays the `stock` instance’s values

Build and run your code, and you’ll see the coffee `Stock` displayed:

```
Coffee: price 4.8
```

You might notice that the price is just one decimal place. I showed you how to provide formatting for your types in [Chapter 9](#).

Now, the dot operator allows you to change values as well as read them. For example, you can alter an instance’s `name` and `last_price`:

```
int main()
{
    stock_prices::Stock coffee{"Coffee", 4.8, 0.0113}; ❶

    std::cout << coffee.name ❷
               << ": price " << coffee.last_price ❷
               << '\n';

    coffee.name = "Tea"; ❸
}
```

```

        coffee.last_price = -19.2; ❸

        std::cout << coffee.name ❹
        << ": price " << coffee.last_price ❹
        << '\n';
    }

```

- ❶ Creates an instance of `Stock` with specific values, as before
- ❷ Displays the `stock` instance's values, as before
- ❸ Alters the member variables
- ❹ Displays the updated values

When you build and run this, you will see the original values followed by the updated values:

```

Coffee: price 4.8
Tea: price -19.2

```

Using a simple `struct` is helpful if you want to group together some simple data. You can change any of the member variables as often as you want. Sometimes, though, you might want to make sure that the original values do *not* change. For instance, changing “Coffee” to “Tea” might cause confusion or annoyance! You do want to be able to change the price, though.

To do that, you can declare a `const Stock`:

```

const stock_prices::Stock coffee{"Coffee", 4.8, 0.0113};

```

You can no longer change any of the values, so the following code won't compile:

```

coffee.name = "Tea";
coffee.last_price = -19.2;

```

You met `const` back in “[Declaring Variables](#)” on page 15, so you might recall that using it expresses an intention to keep the values constant. You can still read the values from a `const Stock`, though, because reading doesn't change the values:

```

std::cout << coffee.name
<< ": price " << coffee.last_price
<< '\n';

```

However, you might want *some* values to change sometimes. For example, `last_price` could update from time to time.

Let's find out more about classes in the next section, building on the `Stock` `struct` to discover how to control changes to member values.

Private and Public Access Specifiers

By default, you can change stock's member variables. You can restrict access to them, setting them away in a section, by marking them as private:

```
struct Stock
{
    private: ❶
        std::string name{};
        double last_price{};
        double volatility{};
};
```

❶ Declares the following members Private

When members are marked as private, only the Stock instance can access their values.

You can make members of your struct accessible to outside code using the keyword public. private and public are two of the three *access specifiers*: members can be given private, public, or protected access. The protected keyword allows related classes to access members, but you can avoid protected member access for most cases. You will learn about related classes in [Chapter 12](#). [Figure 10-1](#) illustrates the difference between private and public access.

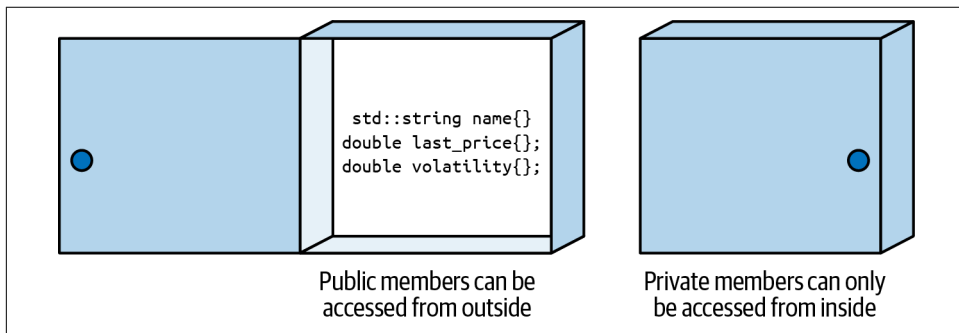


Figure 10-1. Public members are visible to any code, whereas private members can only be accessed from inside

You can still create a Stock instance:

```
Stock stock{};
```

However, you can no longer set the member variables, since they are private. You can't read them either, so you can't display the stock's values.

By default, the struct has public access, so you can get and set the member values directly. Once you introduced the private specifier, however, you lost the ability to access these from outside the struct.

C++ has the keyword `class`, too. By default, a class has private access. If you change the struct to a class, you have an equivalent type and don't need to state that the members are private:

```
class Stock
{
    std::string name{}; ❶
    double last_price{}; ❶
    double volatility{}; ❶
};
```

❶ Declares private members, since no other access specifier is given

A class can have functions as well as variables. To give the private members values, you *could* add public member functions for each member variable, allowing outside code to get or set the values. People often do this, so it's worth knowing how, but I'll show you a better approach in the next section. [Example 10-1](#) demonstrates this for the name, but keep in mind that it is not recommended.

Example 10-1. Getter and setter (not recommended)

```
#pragma once

#include <string>

namespace stock_prices
{
    class Stock
    {
        std::string name{}; ❶
        double last_price{};
        double volatility{};

    public: ❷
        std::string get_name() const ❸
        {
            return name;
        }
        void set_name(const std::string & new_name) ❹
        {
            name = new_name;
        }
    };
}
```

- ❶ Declares what follows as private, until the access modifier
- ❷ Declares what follows as public, until another access modifier
- ❸ Returns the private name
- ❹ Updates the name

These new public functions allow code outside `Stock` to access the private member variables indirectly. The getter is `const`, because it does not change anything; the setter is not `const`, because it *does* change a member variable. You use the dot operator to call them:

```
Stock coffee{};
coffee.set_name("coffee"); ❶
std::cout << coffee.get_name() << '\n'; ❷
```

- ❶ Calls `coffee`'s `set_name` function
- ❷ Calls `coffee`'s `get_name` function

If `coffee` is `const`, you can only call `get_name`, because that function is `const`. Calling `set_name` will give a compiler error. (It *is* possible to set the name for a `const` object once you learn more about classes.)

So why make variables private and provide public getters and setters? The code was simpler when everything was public.

Now, I've introduced the idea of `private` so you can avoid having values being changed from outside. In particular, the name of an instance of `Stock` has no reason to change. The `last_price` and `volatility` might change, but any change will happen as the `Stock` price changes, so these shouldn't be changed for any other reason. By making them private, you have *encapsulated* the data. That means the data and related methods are bundled together, which can make the code more flexible. For example, if everything is public, changing a variable name will ripple through your whole codebase. Since a private variable can be accessed only in the class itself, though, changing variable names has less impact. You also have control over what is allowed to change values.

Let's look at a sensible way to initialize the member variables when an instance is created.

Constructors and Destructors

You can add a member function, called a *constructor*. The constructor is called when an instance is created or constructed. It is called indirectly, without you using the dot operator. The constructor's name matches the class name and can take various parameters. By default, C++ generates a constructor that takes no parameters, which is called an *implicitly defined default constructor*.

You could explicitly define it yourself, adding this function inside `Stock`:

```
Stock()
{
}
```

You can also explicitly request a default constructor like this:

```
Stock() = default;
```

You don't need to do either if you have no other constructors. The implicitly defined default will do the right thing: you relied on this in [Example 10-1](#). If you don't need to add any code to your default constructor, use the `= default` approach. It's less to read and type, and it makes clear that you want a default constructor. Even better, you might not need to write *either* version, since the implicitly defined default will be available to you.

Now, you want to set the member variables, so your constructor can take parameters. This means you will no longer get the default constructor, which is OK for your `Stock` class. If you did need a default constructor, you could use `Stock() = default`, but you want to set sensible values for the member variables. You can add a constructor with a signature providing the values, like this:

```
Stock(const std::string & name, double last_price, double volatility);
```

Let's think about how to implement this new constructor. First, I'll show you what not to do. You might be tempted to set the values inside the braces, as shown in [Example 10-2](#).

Example 10-2. An inefficient way to initialize member variables (not recommended)

```
Stock(const std::string & stock_name,
      double start_price,
      double start_volatility)
{
    name = stock_name;
    last_price = start_price;
    volatility = start_volatility;
}
```

This works, but C++ provides a better approach. Before the opening brace, any data members with initializers are initialized. Did you notice the {} following each data member in the definition of `Stock`? That initializes the data members. If you put the assignments inside the braces, the members are given values a second time. That's not a disaster, but it is wasteful.

You can initialize them just once if you use a *member initializer list*: a list of values to initialize members. This list comes after the constructor's closing parenthesis,), for any parameters, and before the opening brace, {, of the definition. You introduce it with a colon, followed by a comma-separated list of members.

For your `Stock` class, you would do this:

```
Stock(const std::string & stock_name,  
      double start_price,  
      double start_volatility)  
: ❶  
  name(stock_name), ❷  
  last_price(start_price), ❷  
  volatility(start_volatility) ❷  
{  
}
```

- ❶ The colon introduces a member initializer list: the best way to initialize members with the provided values
- ❷ Uses the provided values to set data members

Now each member is assigned only once. I strongly recommend using a member initializer list.

Once you provide a constructor, C++ no longer creates the default constructor for you. That means you cannot create a `Stock` without values anymore. The following will no longer compile:

```
Stock coffee{};
```

If you try to compile it, you'll get various error messages along the lines of "no matching function call to `Stock::Stock`." The compiler is looking for a constructor function called `Stock` inside your `Stock` class and can't find a matching overload. That's a good thing. You want code to provide values explicitly to make a `Stock` instance. Compiler error messages can be intimidating, but it's easier to fix an error at compile time than to fix mysterious behavior at runtime.

You can create a stock with values, though, such as:

```
Stock coffee{"Coffee", 4.8, 0.0113};
```

In addition to a constructor, C++ allows you to have a *destructor*: a function that is implicitly called when an object goes out of scope. In fact, as with the constructor, if you don't provide one, one will be provided for you. Destruction sounds aggressive, but think of it as tidying up. For example, the `std::vector` creates and manages elements for you, so its destructor will clear up the elements. The `std::string` is similar. “How Does a `std::string` Work?” on page 229 showed you how strings work in depth and explains what the destructor does and why. You also saw how files close when a file object goes out of scope: the file's destructor closes the file, so you don't have to remember to do that. The destructor provides a specific place to tidy up if needed. I will show you how important and useful constructors and destructors are in “Resource acquisition is initialization” on page 237.

For the `Stock` class, using the provided destructor is fine. It's worth being aware of the syntax, though, even though you don't need it. To declare a destructor, you use the class name, like you did for a constructor, but add a tilde at the start:

```
~Stock()  
{  
}
```

You can't pass any parameters: the destructor is called for you when your object instance goes out of scope. You can explicitly request the default, as you did for a constructor:

```
~Stock() = default;
```

However, you might not need to do this.

Why might you want code in a destructor? Ideally, you don't need it. If you have member variables that do the right thing in their own destructors, the default destructor will call these for you, so you won't need to do anything. In this case, `Stock`'s name gets destroyed by the `std::string`'s destructor, so the default just works.

To recap, the whole `stock` class looks like this:

```
#pragma once  
  
#include <string>  
  
namespace stock_prices  
{  
    class Stock  
    {  
        std::string name{};  
        double last_price{};  
        double volatility{};  
  
    public:  
        Stock(const std::string & stock_name,  
              double start_price,
```

```

        double start_volatility) ❶
    : ❷
      name(stock_name), ❷
      last_price(start_price), ❷
      volatility(start_volatility) ❷
    {
    }

    std::string get_name() const ❸
    {
        return name;
    }
};

```

- ❶ Defines a constructor
- ❷ Introduces a member initializer list, using the values to set member variables
- ❸ Defines a getter for the name

The `Stock` class has a constructor that takes a name, price, and volatility. These are used in the member initializer list to set member variables. You no longer get a default constructor, since you have provided a different constructor. The class provides a `get_name` member function that returns the name. This is marked as `const` because it doesn't change any member variables. You no longer have the `set_name` function, because the name is set once in the constructor—calling code can no longer change it. Your class also has an implicitly declared destructor.

Take a pause. You've learned a lot. C++ is giving you precise control, which is a good thing. With practice, you'll get a feeling for how to design classes. The compiler will give you errors if you remove a default constructor when you need one, if you call a non-`const` member function for a `const` object, or if you make another mistake.

At the moment, you can't do much more than create instances and report their names. Let's use the `Stock` class and add a method to get the latest price. You know several ways to generate prices. In the next section, you will add a method to generate a price one way. I'll show you some different strategies in [Chapter 12](#).

Using the Stock Class in a `std::vector`

You have a `get_name` function that returns the name of a `Stock`. Let's add a `next_price` function to update the price when this function is called.

In this section, you will use some of the logic from a `get_prices` function you wrote previously, in [Example 7-3](#). This uses the `std::normal_distribution`, seeded with a `std::random_device`. For now, you can add a `std::normal_distribution` as a

member variable, along with an engine, and seed the engine in the constructor. In [Chapter 12](#), you will see ways to build related classes, giving you more flexibility and making testing easier.

Add the new function, shown in [Example 10-3](#), in your header.

Example 10-3. Adding behavior to your class

```
#pragma once

#include <random> ❶
#include <string>

namespace stock_prices
{
    class Stock
    {
        std::string name{};
        double last_price{};
        double volatility{};
        std::default_random_engine gen{std::random_device{}()}; ❷
        std::normal_distribution<double> distrib{}; ❸

    public:
        Stock(const std::string & stock_name,
              double start_price,
              double start_volatility)
            : name(stock_name),
              last_price(start_price),
              volatility(start_volatility)
        {
        }
        std::string get_name() const
        {
            return name;
        }
        double next_price() ❹
        {
            double percent = volatility * distrib(gen);
            last_price += last_price * percent;
            return last_price;
        }
    };
}
```

❶ Includes the random header

❷ Adds a `std::default_random_engine` member variable

- ❸ Adds a `std::normal_distribution` member variable
- ❹ Generates a new price when called

Notice you have two more member variables that you initialize in place:

```
std::default_random_engine gen{std::random_device{}()};  
std::normal_distribution<double> distrib{};
```

The `distrib` relies on the default parameters of `std::normal_distribution`. The generator, `gen`, takes a seed from a call to a `std::random_device`. The new function, `next_price`, uses these to generate a new price.

Create a new *main.cpp* file and make a vector of `Stocks`, as shown in [Example 10-4](#).

Example 10-4. A vector of stocks from an initializer list

```
#include <iostream>  
#include <vector> ❶  
  
#include "stock.h"  
  
int main()  
{  
    using namespace stock_prices; ❷  
    std::vector<Stock> stocks { ❸  
        Stock{"Coffee", 4.8, 0.0113},  
        Stock{"Tea", 171.68, 0.023},  
        Stock{"Sugar", 17.91, 0.05}  
    };  
  
    for(auto & stock: stocks) ❹  
    {  
        std::cout  
            << stock.get_name()  
            << ": "  
            << stock.next_price()  
            << '\n';  
    }  
}
```

- ❶ Includes `vector`
- ❷ States you are using the `stock_prices` namespace
- ❸ Defines a vector of `Stock`
- ❹ Displays each element's `get_name` and `next_price`

If you build and run your code, you will see Stock names and prices:

```
Coffee: 4.75029
Tea: 170.505
Sugar: 18.6116
```

Before you continue, I want you to notice an important point: the range-based for loop doesn't use a const stock:

```
for(auto & stock: stocks)
```

If you change the stock in the loop to const, you will get a compile error, along the lines of:

```
error: passing 'const stock_prices::Stock' ... discards qualifiers
```

The const is a *qualifier*: it qualifies each stock in the range-based for loop as const, so you can only call const member functions. The get_name function says it is const, so it's OK, but next_price changes the object, so it cannot be called for a const object. (You saw a const qualifier at the start of this chapter, when I first introduced get_name.)



Qualify member functions as const when they do not change member variables. These functions can be called by const objects. The compiler will tell you if you try to call non-const functions for objects that should not be changed.

Introducing Classes in Depth

You started with a struct with public data members:

```
struct Stock
{
    std::string name{};
    double last_price{};
    double volatility{};
};
```

You created an instance, giving values to each data member:

```
stock_prices::Stock coffee{"Coffee", 4.8, 0.0113};
```

You don't need to provide values for all the members:

```
stock_prices::Stock coffee{"Coffee"};
```

The braces on the other members ensure that they get default values.

The struct has public access by default, so you switched to using a class instead, making the variables private by default. You added a constructor to the class version, taking values for each member variable. The name could then be provided by calling code but couldn't be changed afterward.

Let's look in more depth at constructors and destructors.

Constructors and Destructors in Depth

Let's add some output in the two special member functions:

```
Stock(const std::string & stock_name,  
      double start_price,  
      double start_volatility)  
{  
    name(stock_name),  
    last_price(start_price),  
    volatility(start_volatility)  
}  
  
{  
    std::cout << "Constructed " << name << " instance\n"; ❶  
}  
  
~Stock()  
{  
    std::cout << "Destructed " << name << " instance\n"; ❷  
}
```

❶ Says when an instance is constructed

❷ Says when an instance is destructed

You will now see output when the functions are called. Construct a single Stock in main:

```
int main()  
{  
    using namespace stock_prices;  
    Stock coffee{"Coffee", 4.8, 0.0113};  
}
```

If you build and run this, you will see an instance be created and then destroyed:

```
Constructed Coffee instance  
Destructed Coffee instance
```


You also put some Stock elements in a `std::vector`:

```
int main()
{
    using namespace stock_prices;
    std::vector<Stock> stocks {
        Stock{"Coffee", 4.8, 0.0113},
        Stock{"Tea", 171.68, 0.023},
        Stock{"Sugar", 17.91, 0.05}
    };
}
```

If you build and run this code, you'll see three calls to the constructor but more calls to the destructor:

```
Constructed Coffee instance
Constructed Tea instance
Constructed Sugar instance
Destructed Sugar instance
Destructed Tea instance
Destructed Coffee instance
Destructed Coffee instance
Destructed Tea instance
Destructed Sugar instance
```

You provided three Stock items in an initializer list for the `std::vector`, so it makes sense that there are three constructors. The items are copied from the initializer list to the `std::vector`, giving you twice as many objects as you might expect. In [Chapter 11](#), you'll see ways to avoid this and find out how the copies are being made.

At a high level, the default constructor is one of several special member functions the compiler can provide for you. Copying is another. The next chapter will show you more details about copying and other special member functions. For this introductory chapter, though, it's enough to know that C++ might generate special functions for you in classes.

You can create Stock from values, or from another Stock:

```
Stock coffee{"Coffee", 4.8, 0.0113}; ❶
Stock more_coffee(coffee);           ❷
```

- ❶ Constructs Stock from values
- ❷ Constructs Stock by copying an existing instance

The copying is possible because C++ has provided a default copy constructor.

There is more to learn, but for now, you can define a `class` in various ways. You’ve split declarations and definitions of functions between source and header files several times already, so now let’s see how to do that for a class.

Splitting a Class Between Header and Source Files

You defined everything for your `Stock` class in the header in [Example 10-3](#). But you can put some of your function definitions in a source file instead. This is sensible if you have functions that are longer or more involved. Everything that includes the header will need to recompile when you make any change in your header file. If you have code in a source file instead, changing that avoids the need to recompile other source files when you use a build system or IDE (recall an IDE is an integrated development environment like Visual Studio or CLion).

Take the constructor and `next_price` definitions out of your header, because they’re the slightly longer functions. You need to add their declarations in the class:

```
#pragma once

#include <random>
#include <string>

namespace stock_prices
{
    class Stock
    {
        std::string name{}; ❶
        double last_price{}; ❶
        double volatility{}; ❶
        std::default_random_engine gen{std::random_device{}()}; ❶
        std::normal_distribution<double> distrib{}; ❶

    public:
        Stock(const std::string & stock_name,
              double start_price,
              double start_volatility); ❷

        std::string get_name() const ❸
        {
            return name;
        }

        double next_price(); ❹
    };
}
```

- ❶ Declares private member variables, as before
- ❷ Declares the constructor taking some values, but does not define it
- ❸ Defines the `get_name` function, which is OK in the header, since it's so short
- ❹ Declares the `next_price` function, but does not define it

You need to provide the definitions. Create a file called *stock.cpp*. You will put both definitions in here. They are members of `Stock` and are in the `stock_prices` namespace, so you need to specify `stock_prices::Stock::` at the start of each definition:

```
#include "stock.h"

stock_prices::Stock::Stock( ❶
    const std::string & stock_name,
    double start_price,
    double start_volatility)
    : name(stock_name),
      last_price(start_price),
      volatility(start_volatility),
      gen(std::random_device{}())
{
}

double stock_prices::Stock::next_price() ❷
{
    double percent = volatility * distrib(gen);
    last_price += last_price * percent;
    return last_price;
}
```

- ❶ Defines the constructor for `Stock` in the `stock_prices` namespace
- ❷ Defines the `next_price` function for `Stock` in the `stock_prices` namespace

Now you need to add *stock.cpp* to your build instructions.

If you keep longer functions in source files, regardless of whether they are free functions or class-member functions, this can speed up compile times. If you put everything in a header file, every source file that includes that header will get recompiled when you change the header. **Figure 10-2** shows your stock source and header files, along with *main.cpp*. Updating the *stock.cpp* file will only affect the stock object file, whereas changing the header affects both the stock and main object files.

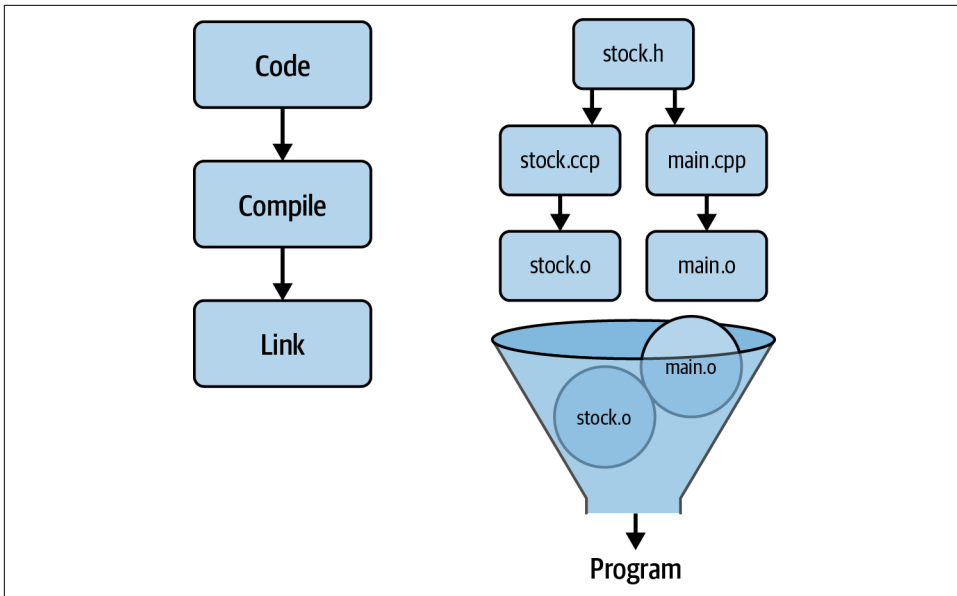


Figure 10-2. Using a source and header for a class

Conclusion

You have created and used your first class. First, you used a `struct` with three data members. You used brace initialization to set these values and the dot operator to access them. Members of a `struct` are public by default. You then used a `class`, which has private members by default. You added a constructor, so data could be set. You also provided a `const` member function, so code outside the `class` could get the `Stock`'s name. You marked this member function as `const` because it does not change member values. A `const` object can only call `const` member functions.

You gave your `Stock` class a default destructor too, printing a message when called. You didn't need to do this, but you saw how it is used when an instance goes out of scope. You then reverted the class to rely on an implicitly declared destructor.

The constructor and destructor are special member functions that the compiler can provide for you. You saw how to copy an instance, which is another special member function. In [Chapter 11](#), you will learn more about copies and other special member functions.

You added behavior to your class, generating the `next_price` in a non-`const` member function. You previously wrote several ways to generate prices, so you're ready to learn how to provide different pricing strategies to your `Stock` class in [Chapter 12](#).

You also learned how to split your class between a source and header files. Try to keep code that might change or longer code in source files. This can improve build times, because the header contains less code. It can also make your code easier to understand: you might not need to know the details of how a `next_price` is generated in order to use the class.

Classes: Special Member Functions and Move Semantics

In [Chapter 10](#), you wrote your own class and discovered that the compiler can generate special member functions for you, such as constructors and destructors. This chapter will show you the other special member functions, when they are used, and why they matter.

This chapter is a deep dive to explain some important ideas, including *move semantics*: a way to make your code more efficient by transferring resources. As it stands, your `Stock` class works well enough, but taking time to understand what is happening will give you a solid C++ foundation.

You will also learn about how a `std::string` works in detail, seeing what happens in each special member function. This will show you why C++ provides various special member functions for classes and what they do.

Copying Objects

In the previous chapter, in [Example 10-4](#), you made a vector of stocks, as shown in [Example 11-1](#).

Example 11-1. A vector of stocks from an initializer list

```
#include <vector>

#include "stock.h"

int main()
{
    using namespace stock_prices;
```

```
std::vector stocks {
    Stock{"Coffee", 4.8, 0.0113},
    Stock{"Tea", 171.68, 0.023},
    Stock{"Sugar", 17.91, 0.05}
};
}
```

This code calls a constructor three times but calls six destructors. The initializer list contains three objects that are copied to the `std::vector`. The copying happens via a special member function called a *copy constructor*, designed to make a copy of an existing object. Figure 11-1 shows how you end up with two objects when you copy, so the original three need to be destroyed along with the three copies.

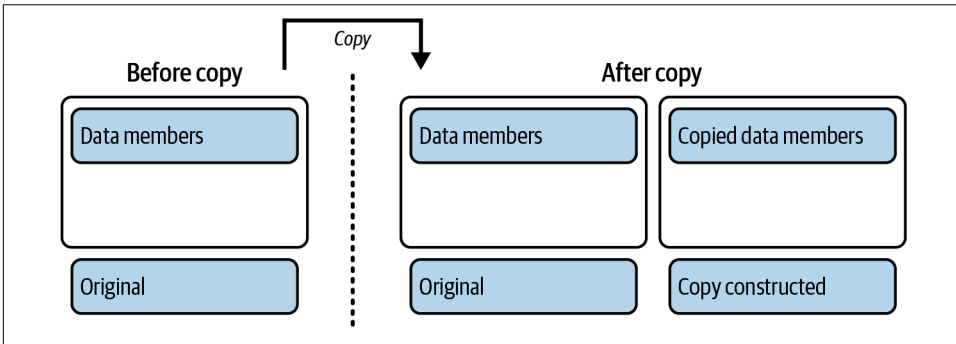


Figure 11-1. Copying an object

Let's look at the details of a copy. The compiler can provide a *copy constructor* for you with this signature:

```
Stock(const Stock & other);
```

Like the other constructors you have used so far, a copy constructor has the name of the class. It takes a single parameter, the *other* or original object you are copying from, and constructs a new instance. The parameter, *other*, is an existing object, so you can take copies of some or all members in this function. You can write your own copy constructor, but you don't need to do that here. The default does what you want.

You only get a default copy constructor under certain circumstances, so it's useful to be able to ask for a copy constructor explicitly. You can request a copy constructor using default:

```
Stock(const Stock & other) = default;
```

You can also disable copying using `delete`:

```
Stock(const Stock & other) = delete;
```

Try disabling copying in your code by adding the `= delete` version to your stock header file.

The `main` code will no longer compile, because the stocks can no longer be copied from the initializer list. Sometimes you'll want to avoid making a copy. If you copy `stock`, you will end up with two objects whose prices will diverge when you call `next_price`. That's not useful—the price of a stock should be one value.

How do you make a vector of `Stock` if you can't copy it? Since C++11, C++ has provided another special member function that allows you to create an object by “moving” another instance. Let's see how.

Moving Objects

Moving an object has a very specific meaning in C++. At a high level, you can have another constructor called a *move constructor*, which is called for temporary objects. You met temporary objects in “[Joining std::strings Efficiently](#)” on page 195, when you learned how temporary strings are made if you join several strings.

Temporary objects can be made in various ways and are often called *rvalues*. In C, *rvalue* means an expression to the right of an equal sign, such as:

```
int number = 1 + 2;
```

C++ borrows the term *rvalue* but uses it more generally, to mean a value with no name. The number to the left of the equal sign is an *lvalue*, while the expression `1 + 2` is an *rvalue*. The sum is temporary, in the sense that it goes out of scope after the statement. A *temporary* (used as a noun) usually means something without a name, such as the result of joining two strings:

```
"Hello, "s + "again"s
```

You can write a function, including a constructor, by taking an *rvalue* by reference. To do this, use `&&` in the signature, like this:

```
Stock(Stock && other);
```

This is called a *move constructor* because it creates a value from an *rvalue* reference. It doesn't really move the other `Stock`, but it provides an overload that takes an *rvalue* reference. This constructor can take over ownership of members in the other `Stock`, rather than making copies, which can be more efficient. Allowing an object to take ownership of another's members is called *move semantics*. You'll see more details in “[Copies and Moves in Depth](#)” on page 229.

You can also mark a member function as `noexcept` if it won't throw an exception. This is common for move constructors, because they don't need to allocate memory to make new elements or do other things that can throw exceptions. As with other special member functions, you can request a default move constructor:

```
Stock(Stock && other) noexcept = default;
```


When you deleted the copy constructor, you signaled to the compiler that you want control over the special member functions, so C++ won't generate the move constructor for you. You'll have to request one or write one yourself if you need it.

Change the code in `main` to use `push_back` to populate the vector:

```
std::vector<Stock> stocks;  
stocks.push_back(Stock{"Coffee", 4.8, 0.0113});
```

Notice that you're making a temporary without a name: `Stock{"Coffee", 4.8, 0.0113}`. Add the default move constructor inside the `Stock` class:

```
Stock(Stock && other) noexcept = default;
```

Now your code will compile.

The `std::vector`'s `push_back` can use the move constructor for the temporary `Stock` to add a new element. First it makes a temporary coffee `Stock` and then uses the move constructor to put it in a vector. As with the copies in [Figure 11-1](#), you end up with two objects. However, the moved object can use the original's data members, as shown in [Figure 11-2](#).

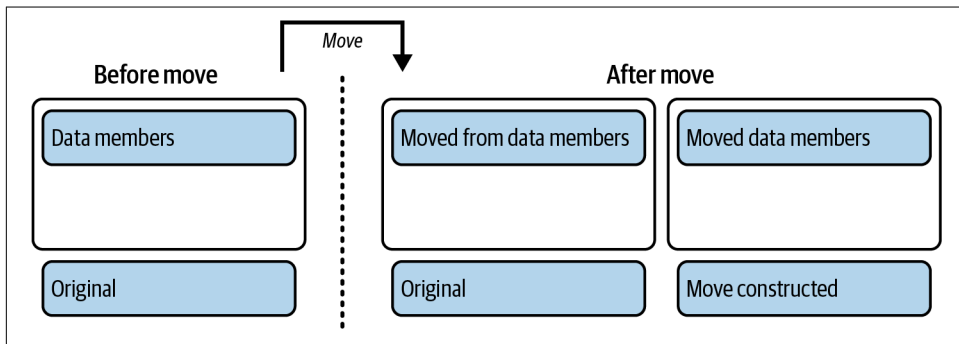


Figure 11-2. Moving an object

Unlike a copy operation, which creates a duplicate of an object, the move operation *transfers* the data to the other object. As a consequence, the original object can be modified in a move operation. Such objects are called *moved-from* objects. They're in a different, unspecified state after the move. However, a moved-from object will still be in a valid state. For example, you can get the name of a moved-from object from `stock`, but it might be a default (empty) string. The transfer, however, is far more efficient than creating a copy of the string.

There are two more special member functions that create objects from others. Let's look at these so you'll know all the special member functions.

Avoiding duplicates with `emplace_back`

Calling `push_back` generated a temporary `Stock` object. You can make a `std::vector` create an object in place, without copies or moves, by using `emplace_back`:

```
stocks.emplace_back("Coffee", 4.8, 0.0113);
```

This uses the parameters to create a `Stock` directly in the `std::vector`, so you can avoid the extra temporary `Stock` objects.

Move and Copy Assignments

In addition to constructing objects from other objects, you can assign existing objects using others. For an `int`, you can start with one value and change to another:

```
int number = 10;  
number = 101; ❶
```

❶ Assigns a new value to `number`

To do this for your objects, you need to add *assignment operators*. You have seen the move and copy constructors: similarly, you can have copy and move *assignments*. You can request defaults:

```
Stock & operator = (const Stock & other) = default;  
Stock & operator = (Stock && other) noexcept = default;
```

Both are called `operator =`. They take different parameters. The copy takes the other `Stock` by `const` reference, and the move takes an rvalue reference and is flagged `noexcept`. Add these to your `stock` class definition.

You've seen that making a copy of a `stock` is a bad idea, but you do need to understand what the copy assignment is used for. Try this code in `main`:

```
Stock original{"Coffee", 4.8, 0.0113};  
Stock copy = original;
```

Even though you can see an equal sign, the code is constructing a copy, so it tries to use the copy constructor. If the copy constructor is deleted, you will see a compiler error along the lines of:

```
error: use of deleted function  
      'stock_prices::Stock::Stock(const stock_prices::Stock&).'
```

Creating the copy tries to use the deleted copy constructor. You can use `=` when you create an object, but I've encouraged you to use the uniform initialization syntax with `{}`, optionally providing a value in the braces. This makes it clearer that you're using a copy constructor:

```
Stock original{"Coffee", 4.8, 0.0113};
Stock copy{original}; ❶
```

❶ Equivalent to `Stock copy = original;` but more explicit

The copy assignment is used when you set an existing instance to another value, like this:

```
Stock coffee{"Coffee", 4.8, 0.0113};
Stock tea{"Tea", 171.68, 0.023};
tea = coffee;
```

To recap, you constructed two objects and then set the second to be a copy of the first—so the copy assignment is used here. Copy assignment changes an existing object, while copy construction creates a new object.

Move assignments, on the other hand, are used for rvalues. You can make an object into an rvalue by calling `std::move`, defined in the `<utility>` header:

```
Stock coffee{"Coffee", 4.8, 0.0113};
Stock moved_coffee{"Coffee", 4.9, 0.0124};
moved_coffee = std::move(coffee); ❶
```

❶ “Moves” the original `coffee` instance into the `moved_coffee` instance

Because `coffee` is named and not a temporary, you need to call `std::move`. Otherwise, you would be calling the copy assignment. The name “Coffee” would then be duplicated. Using the move assignment avoids the duplication. Now `moved_coffee` takes ownership of the name. “Copies and Moves in Depth” on page 229 will explain how string copies and moves work in more detail. For now, note that a move doesn’t really move anything.

Ownership of the `std::string` name in the original `coffee` passes to `moved_coffee`. The name of the original stock after the “move” is unspecified, but it might be an empty string.

Providing moves gives a class *move semantics*. When do you need move semantics? Well, moves can make your code more efficient. When you move an object, you’re saying that you have finished with it, so you can avoid duplicate strings and other resources. If an object doesn’t have move semantics, copies might be made instead, potentially leading to unintentional duplication. If there is no way to copy either and that’s required somewhere, you get a compiler error.

What C++ is doing here is giving you fine-grained control. If your code needs an operator or constructor, let any compiler error guide you.



Ideally, you can avoid needing to specify *any* of the member functions. This is called the *rule of zero*: you provide zero special member functions because the defaults do the right thing.

Let's recap the `Stock` class. You want to avoid copies, because having two stock instances with the same name is confusing. However, the vector needs to be able to move elements, either when they are added or if a reallocation is required. You therefore request the defaults for moves, as shown in [Example 11-2](#).

Example 11-2. Deleted copies and defaulted moves in a class

```
#pragma once

#include <random>
#include <string>

namespace stock_prices
{
    class Stock
    {
        std::string name{};
        double last_price{};
        double volatility{};
        std::default_random_engine gen;
        std::normal_distribution<double> distrib;

    public:
        Stock(const std::string & stock_name,
              double start_price,
              double start_volatility); ❶

        Stock(const Stock & other) = delete; ❷
        Stock(Stock && other) noexcept = default; ❸

        Stock & operator = (const Stock & other) = delete; ❹
        Stock & operator = (Stock && other) noexcept = default; ❺

        std::string get_name() const ❻
        {
            return name;
        }

        double next_price(); ❼
    };
}
```

- ❶ Declares the constructor, which is defined in *stock.cpp* from the previous chapter

- ❷ Deletes the copy constructor
- ❸ Lets the compiler generate the move constructor
- ❹ Deletes the copy assignment operator
- ❺ Lets the compiler generate the move assignment operator
- ❻ Returns the stock name, as before
- ❼ Declares the `next_price` member function, which is defined in the `stock.cpp` file

You can now put `stock` in a vector using `push_back`:

```
#include <iostream>
#include <vector>

#include "stock.h"

int main()
{
    using namespace stock_prices;
    std::vector<Stock> stocks;
    stocks.push_back(Stock{"Coffee", 4.8, 0.0113}); ❶

    for(auto & stock : stocks)
    {
        std::cout << stock.get_name() << ": " << stock.next_price() << '\n';
    }
}
```

- ❶ Pushes back a temporary `Stock`, using the move constructor to put this in the vector

You've covered a lot of ground, and now you know much more C++. In general, you can often use the rule of zero, meaning you don't need to start defaulting or deleting special member functions. However, you don't want copies of `stock` for your trading game, so you deleted the copies.

As soon as you delete or write your own special member functions, the rule of zero no longer applies. In this situation, you can add `default` or `delete` to any special member function to control what you get, or you can write your own.

You need to think about five things: copy and move constructors, copy and move assignments, and the destructor. Thus, the *rule of five*: you should be explicit about these five special member functions. For completeness, let's look at what's happening under the hood when you use copy and move functions.

Copies and Moves in Depth

The default moves and copies for the `Stock` class need to deal with the class members: one `string` and two `doubles`. `Doubles` are built-in types, so they can always be copied. Moves are relevant when an object owns a resource. For example, if a `std::string` manages a message for you, it can be moved and copied. The defaults for your class rely on the `std::string` functions. Let's take a look.

How Does a `std::string` Work?

Understanding how the `std::string` works sheds light on the inner workings of any object that owns a resource, including a `std::vector`. This gives you a deeper knowledge of construction, moves, copies, and destructors for any C++ object.

Let's consider what happens when you declare an `int` and a `std::string`:

```
int main()
{
    int number = 42;
    std::string message{"Tea"};
}
```

You have declared two variables on the *stack*: a place in memory for local variables and function calls. In [Chapter 2](#), you learned that when a function is over, local variables are no longer available because they have gone out of scope. So, inside `main`, the stack has a `number` and a `std::string`.

Built-in types, like `int`, have a fixed size, but the `std::string` and `std::vector` can change size dynamically. They need somewhere to store their elements. For example, where does the string “Tea” go? For objects that vary in size at runtime, like a `std::vector` or `std::string`, the answer is (often) the *heap*. The heap is another part of memory that can be “allocated” on demand. Objects can request heap memory, but an exception might be thrown if none is available.

The `std::string` “Tea” needs space for three characters plus a null terminator. The `std::string` constructor can allocate space for the characters by requesting four contiguous `chars` from the heap. (For smaller strings, you often find the `std::string` has a small, fixed-sized array inside instead, called the *small string optimization*. This is quicker than allocating memory on the heap.) C++ provides an operator, `new`, to allocate heap memory, but you'll almost never need to use this operator yourself. The `std::string`'s constructor does it for you. When the `std::string` goes out of scope, the destructor is called. It releases the heap memory, if it used some, via a call to the operator `delete`. Again, you almost never need to use this yourself, since the `std::string`'s destructor does the call for you.

Figure 11-3 illustrates what happens when a string is created and destroyed. Both the number and message are on the stack. They are local to the `main` function. The `int` is on the stack. The string object is on the stack, too, but it needs to put “Tea” somewhere. Its constructor requests heap memory, which may have some leftover data from previous uses. The characters are then put on the heap. When the string goes out of scope, the destructor is called, which lets go of the heap memory. Other objects can then use that memory.

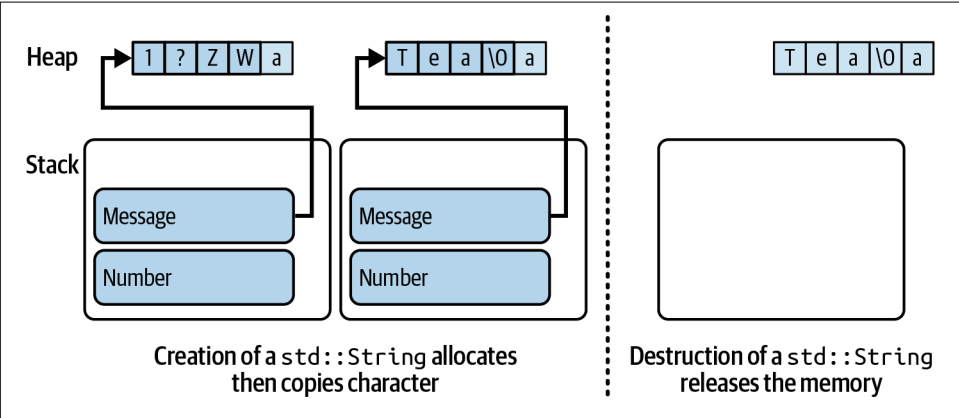


Figure 11-3. Creating a string allocates memory on the heap

When you copy a string, you want a duplicate, also called a *deep copy*: a distinct, but initially identical, object. A copied string can change independently of the original. The copy allocates extra memory and duplicates each character, as shown in Figure 11-3. Both strings point to somewhere on the heap, and both need to deallocate memory when they are destroyed, as shown in Figure 11-4.

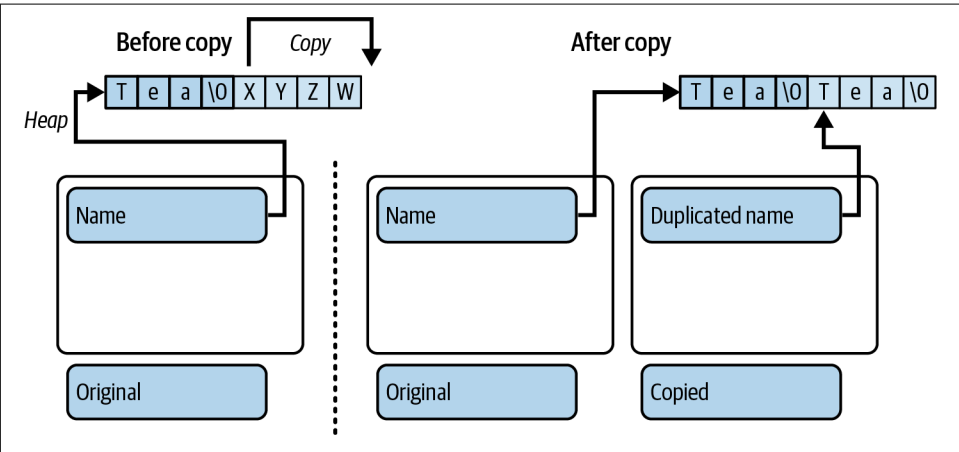


Figure 11-4. Copying a string allocates memory on the heap

A move, however, does *not* need to allocate new memory on the heap: the moved-from string transfers ownership to the new string object. The old string is in an indeterminate state and could be empty. **Figure 11-5** shows the move transferring ownership of the heap-allocated characters, which means “Tea” is no longer duplicated. The arrow shows that the moved-to string now owns the underlying memory on the heap and is responsible for deallocating it.

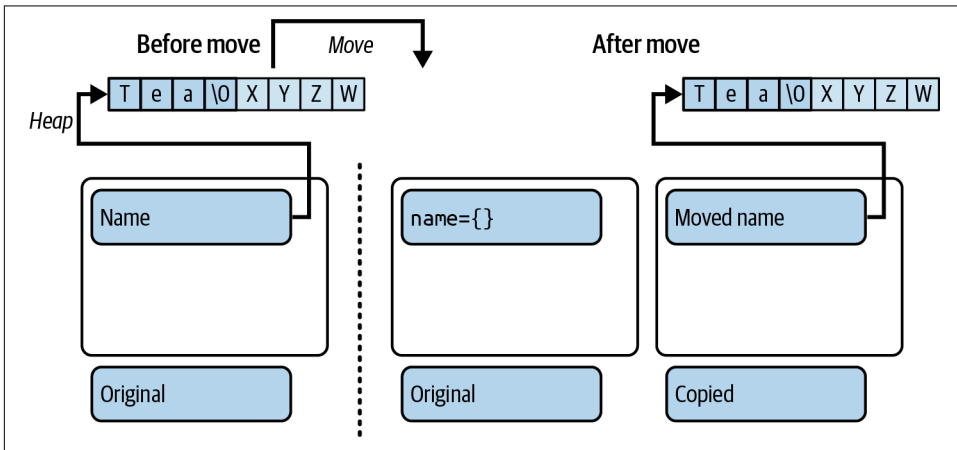


Figure 11-5. Moving a string does not allocate memory on the heap

Knowing what is happening with a `std::string` means you can reason through what is happening with your `Stock` class. Let’s take a look.

Move Constructors and Move Assignments

Let’s consider the move constructor first. A moved stock can take ownership of another stock’s name using `std::move`. Because the other name is not a temporary, you need to call `std::move`—otherwise, you would be calling the copy assignment. The numeric values can just be copied: there’s no data to take ownership of. The move constructor therefore looks like this:

```
Stock(Stock && other) noexcept ❶
    : name{std::move(other.name)}, ❷
      last_price{other.last_price}, ❸
      volatility{other.volatility} ❸
{
}
```

- ❶ Uses `&&` and `noexcept`
- ❷ Transfers ownership of the name

❸ Copies the numeric values

The move assignment has similar requirements but needs more code. It can't use the member initializer list, since member initializer lists can be used only in constructors. Also, in general, move assignments should check for *self-assignment* (when `other` is the current object). Self-assignment might leave the object in an undefined state, so it's best avoided. You don't need to do the extra work to reassign members to themselves.

The move assignment also needs to return a `Stock` reference. You use the keyword `this` to represent a pointer to the current object, so the code can compare the addresses of this object and the other object. The `&` operator returns an object's address. You use `*this` to dereference the `this` pointer, giving the return value you need:

```
Stock & operator = (Stock && other) noexcept
{
    if (&this != &other) ❶
    {
        name = std::move(other.name); ❷
        last_price = other.last_price; ❸
        volatility = other.volatility; ❸
    }
    return *this; ❹
}
```

❶ Checks that the object is not being moved to itself

❷ Transfers ownership of the `name`

❸ Copies the numeric values

❹ Returns the current object

The defaults for both move member functions do the right thing, so you don't need to add the code. Knowing what is likely to be generated helps you understand what is happening under the hood.

Copy Constructors and Copy Assignments

You probably don't want to copy a `Stock`, but it's worth thinking through how the copy member functions might be implemented. Currently, they are deleted:

```
Stock(const Stock & other) = delete;
Stock & operator = (const Stock & other) = delete;
```

You can ask for defaults if you want copies, but for practice, let's implement them.

The copy constructor is similar to the move constructor:

```
Stock(Stock & other)
    : name{other.name}, ❶
      last_price{other.last_price}, ❷
      volatility{other.volatility} ❷
{
}
```

❶ Copies the name, rather than moving it

❷ Copies the numeric values

The work happens in the member initializer list, so the function body is empty. The `name` is copy-constructed from the `other` `name`, because the `std::string` supports the deep copy you need.

The copy assignment can't use the member initializer list, since it's not a constructor. It also needs to return the copied object using `*this`, as you saw in the move assignment:

```
Stock & operator = (Stock & other)
{
    name = other.name; ❶
    last_price = other.last_price; ❷
    volatility = other.volatility; ❷
    return *this; ❸
}
```

❶ Copies the other name

❷ Copies the numeric values

❸ Returns the current object

You don't usually need to implement these special functions yourself, but looking at the implementations demonstrates how the functions might be implemented. " The important thing to remember is that making a copy does not change the original—but a move might. For example, the copy duplicates the `name` but “steals” the original's `name`, potentially leaving it empty.

Conclusion

You have met six special member functions for classes:

- Default constructor
- Destructor

- Copy constructor
- Copy assignment
- Move constructor
- Move assignment

These can be provided for you, or you can request them using `default`. You can remove them using `delete`. Often, you won't need to declare these yourself. When they do the right thing, you are using the rule of zero. You saw that deleting the copies stops the moves from being provided. Once you define or delete copies or moves or implement your own destructor, you should be using the rule of five and thinking about copies, moves, and the destructor.

In [Chapter 10](#), you saw copies being made from an initializer list provided to a `std::vector` via a generated copy constructor. This chapter showed you the difference between copies and moves and how these functions might be implemented for you. You looked in detail at how a `std::string` works, seeing how memory on the heap is allocated in construction and released by the destructor.

Providing move constructors and move assignments gives a class move semantics. The “move” doesn't really move anything—it transfers ownership. Move semantics can make your code more efficient when you learn how to use temporary and finished-with objects effectively. The move functions take an rvalue reference, using `&&`, and should be `noexcept`. You can use `&&` for any function. A move will leave the moved-from object in an unspecified—but valid—state.

Providing a user-declared destructor, copy-constructor, or copy-assignment operator stops the move constructor and move assignment from being provided. Either use the rule of zero, avoid adding any of the special member functions, or be explicit about which functions you do and don't want.

You learned to use the copy constructor when you construct an object from another, whether you use the equal sign or braces. The following are therefore equivalent:

```
Stock original{"Coffee", 4.8, 0.0113};
Stock first_copy = original;
Stock second_copy{original};
```

The brace initialization is clearer.

You also met the keyword `this`, which refers to the current object. The move assignment and copy assignment both return `*this`, so dereference the `this` pointer to return a reference to the current object.

The next chapter shows you more about using the heap. You will then be able to vary the behavior of your stocks for a bigger trading game in [Chapter 12](#).

Memory Management with `std::unique_ptr`

This chapter shows you how to work effectively with objects on the heap to ensure that memory is released when you've finished with it. You met `char` pointers in [Chapter 9](#) and saw more details on the `std::string` in [Chapter 11](#). The `std::string` provides a higher level of abstraction than `char` pointers, making your life easier. Like the `std::vector`, the `std::string` resizes for you, and both tidy up when they go out of scope, so you don't need to do it yourself. C++ provides other features to help you clean up when you're done with an object. If you do want cleanup objects allocated on the heap, you can avoid the low-level, manual manage memory by using a *smart pointer*: a type that works like a pointer but tidies up for you. There are several types of smart pointer, so let's start with the easiest to use. You will then be able to extend your trading game in [Chapter 13](#) using a smart pointer.

So why would you *want* to use the heap? So far, you haven't needed to do this directly yourself, even though `std::string` and `std::vector` might use the heap in the background. [Chapter 13](#) will show you an important use case for heap objects: allowing behavior to vary with class type. Another use case is for objects that change size. Many objects have a fixed size, such as integers. In contrast, the `std::vector` and `std::string` can both contain varying numbers of elements, so use the heap to allocate *dynamically* when you don't know the required size at compile time.

This chapter will show you how to handle pointers to dynamic memory in a smart way that keeps your code safe. You'll learn more about constructors and see a new way to test code at compile time. You will also revise references, and by the end of this chapter you'll be ready to write an improved trading game in the next chapter. Since [Chapter 10](#) you have been learning to use building blocks to create, move, and copy classes, and this chapter will get you ready to use classes to vary behavior.

Creating a `std::unique_ptr`

The `<memory>` header provides various smart pointers and functions to create them. I will show you how to use a `std::unique_ptr` in this section, and then “[Smart Pointers in Depth](#)” on page 239 will give a brief overview of other types of smart pointer. Smart pointers are easier to use than raw pointers, because they handle memory management for you. When you acquire heap memory, you should release it when you are finished. Otherwise, your program will hold on to the memory, which could lead to your system running out of memory. You can use smart pointers to handle other resources too.

The `std::unique_ptr` goes hand in hand with `std::make_unique`. The `std::make_unique` function calls `new` for you, and the smart pointer’s destructor calls `delete` by default. (I mentioned these functions to you in “[How Does a `std::string` Work?](#)” on page 229.) If you use smart pointers, you don’t need to deal with these functions directly.

Make a new *main.cpp* file. Include the `memory` header and your `Stock` header:

```
#include <memory> ❶

#include "Stock.h" ❷

int main()
{
    using namespace stock_prices; ❸
    auto asset{ ❹
        std::make_unique<Stock>("Coffee", 4.8, 0.0113) ❺
    };
}
```

- ❶ Includes the `memory` header
- ❷ Includes your `Stock` header
- ❸ Uses the `stock_prices` namespace
- ❹ Declares an `asset`, which is a `std::unique_ptr<Stock>`
- ❺ Creates the object on the heap for you

`std::make_unique` is a function template. You specify what type of `std::unique_ptr` you want in the angle brackets, `<>`, and provide constructor arguments as parameters. The function returns a `std::unique_ptr<Stock>`.

Because you are requesting heap memory, you might get an exception thrown if none is available. If all goes well and you get the requested memory, a `Stock` object is

constructed for you on the heap. You should release the memory when you're done. Otherwise your program will keep hold of it until the program ends. This is called a *memory leak*: holding on to a resource for longer than needed.

You also need the object's destructor to be called. The `delete` keyword helpfully does both for you. When the `asset` pointer goes out of scope at the closing brace of the `main` function, the smart pointer, by default, calls `delete` so it calls the object's destructor and releases memory back to the heap. Figure 12-1 shows how this happens.

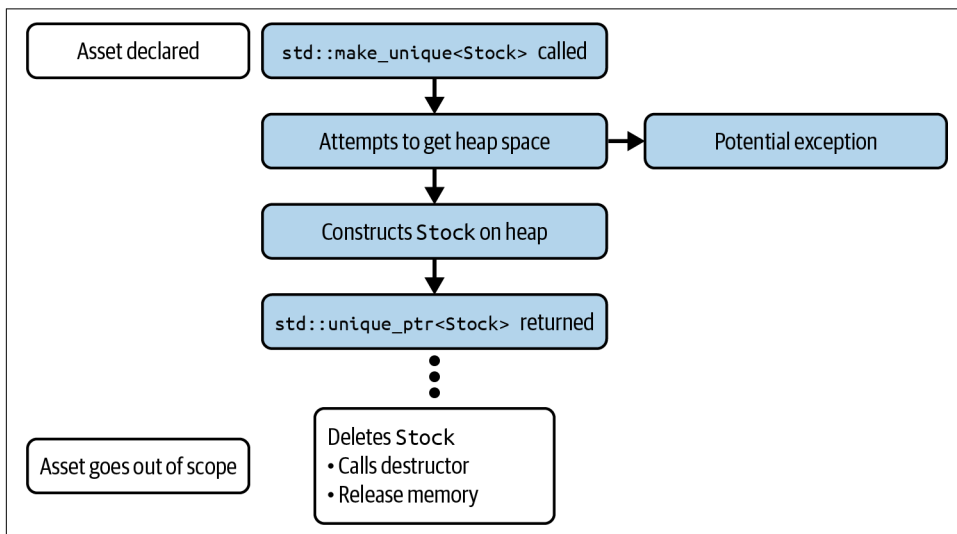


Figure 12-1. Creating a smart pointer with `make_unique`

You don't need to remember all these steps, fortunately. The `asset` smart pointer handles the details for you.

Resource acquisition is initialization

This C++ pattern of automatically doing something to tidy up in a destructor is called “Resource Acquisition Is Initialization,” or RAII. RAII is often regarded as the most important idiom of C++; all resources should be handled by RAII. It is not the most memorable name, but RAII means that you acquire something, often on construction, and then automatically release it in the destructor. You saw how files automatically close for you when they go out of scope—that’s another example of RAII. The `std::vector` and `std::string` also tidy up in the destructor.

The `std::unique_ptr` handles a pointer with exclusive access. Its constructor takes ownership of the pointer, so it can delete the pointer in its destructor. If you had two copies of the same underlying pointer, you'd need to decide which should own the pointer and therefore be responsible for tidying up. You can move a unique pointer or hand ownership of it to another smart pointer. Use a *unique pointer*, `std::unique_ptr`, as your first choice of smart pointer. They are the simplest to use and have the least overhead.

Using a `std::unique_ptr`

Your `asset` points to a `Stock`, so the code doesn't call the member functions directly. You can't do much with pointers, apart from pointing them to something else, trying to delete them, or looking inside them to see what they point to (recall that this is called *dereferencing a pointer*). You have done this before, when you used operator `*` to dereference this in [“Move Constructors and Move Assignments” on page 231](#).

When you dereference a pointer, you get the value it points to. Dereferencing gives you access to the object itself, so you can call member variables or functions of that object. The same goes for smart pointers. For instance, you could call `(*asset)` first and then use the dot operator to use member functions:

```
(*asset).get_name();
```

That's a lot to remember, so C++ allows you to use operator `->` as shorthand:

```
std::cout << asset->get_name() << ": " << asset->next_price() << '\n';
```

Figure 12-2 shows the smart pointer pointing to somewhere on the heap. Think of the operator `->` as a way to point inside to the member functions. This operator is also sometimes called the *arrow operator* because it looks like an arrow.

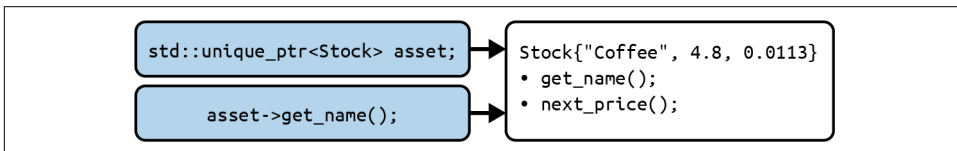


Figure 12-2. A *unique pointer* to an object on the heap

You can also reset the pointer, either to a new pointer or to a sentinel value called `nullptr`, which means the smart pointer is no longer pointing at anything. Since this is possible, the `std::unique_ptr` provides an operator `bool` that allows you to check if there is anything to dereference. As with other types that support operator `bool`, you can then check that the object is usable with an `if`:

```
if(asset)
{
}
```

You are all set to use unique pointers in [Chapter 13](#), but the rest of this chapter will give you some more detail first, for a fuller understanding.

Smart Pointers in Depth

Smart pointers provide a safe way to work with objects on the heap. The alternative to a smart pointer is a *raw pointer*, which is a memory address to the stack or heap, without the smarts. They are harder to work with than smart pointers, since you need to remember to delete them yourself. The raw pointer just points and leaves you to tidy up.

Sticking with smart pointers makes your life simpler, but being able to recognize raw pointers is useful.

That said, let's play a little with pointers and references. This slight digression will help you understand them better. You will also get further practice with references.

More on Pointers and References

Consider the following code:

```
int value{42}; ❶
int * pointer_to_value = &value; ❷
int & reference_to_value = value; ❸
std::cout << "value " << value
    << ", pointer " << *pointer_to_value
    << ", reference " << reference_to_value << '\n';
```

- ❶ Defines an integer on the stack
- ❷ Defines a pointer to the integer, finding value's address using &
- ❸ Defines a reference to the integer

You can put the code in a `main` function in a file and build it without my help now.

You get the address of an object using operator `&`, which you used in “[Move Constructors and Move Assignments](#)” on page 231. The `<memory>` header also includes the `std::addressof`. You could use that instead, but you will often see people using the `&`. You can store an object's address in a pointer and use operator `*` to dereference the pointer, obtaining the value. You don't need to use operator `->` for a built-in type like an `int`, since the dereference returns the object's value. You only use the arrow operator to access class members and functions.

You can also take a reference to the value, declaring an `int &`. The reference gives you another way to refer directly to the value. The reference is an *alias*, or nickname, for an existing object, so it can't be invalid. In contrast, pointers can point anywhere,

including invalid memory, which makes them more difficult to use safely. You should at least check that a pointer is not `nullptr` before using it:

```
if(pointer_to_value) ❶  
{  
}
```

❶ Checks the pointer isn't `nullptr`

If the pointer is null, you can't (safely) dereference it: if you do, you'll get undefined behavior. However, a pointer can pass such a check and still be invalid. For example, it may be pointing to heap memory that's been released. Raw pointers are hard to use!

The reference and pointer both see the value, so the code outputs the following:

```
value 42, pointer 42, reference 42
```

The reference and pointer can also see changes to the value. For example, let's say you change the value:

```
value = 51; ❶  
std::cout << "value " << value  
    << ", pointer " << *pointer_to_value  
    << ", reference " << reference_to_value << '\n';
```

❶ Changes the value

All three variables display the change:

```
value 51, pointer 51, reference 51
```

You can make the pointer point to a different `int`:

```
int another_value{-5}; ❶  
pointer_to_value = &another_value; ❷  
std::cout << "value " << value  
    << ", pointer " << *pointer_to_value  
    << ", reference " << reference_to_value << '\n';
```

❶ Declares another `int`

❷ Changes the pointer to point at `another_value`

The value and reference remain unchanged, while the pointer now uses the other value:

```
value 51, pointer -5, reference 51
```

So, you can switch the pointer to point to a different value. However, you can't switch the reference. Changing the value of the reference changes the original value and anything pointing to it. Reset the pointer to point to `value`'s address and try changing the reference:

```

pointer_to_value = &value; ❶
reference_to_value = 101; ❷
std::cout << "value " << value
    << ", pointer " << *pointer_to_value
    << ", reference " << reference_to_value << '\n';

```

❶ Resets the pointer to original value's address

❷ Changes the reference

All three are updated:

```
value 101, pointer 101, reference 101
```

Remember, a reference always refers to the same object, while a pointer can be switched to point elsewhere.

Thinking through what happens with an `int` is always a good place to start when you want to practice. Now that you have a solid grounding in references and pointers, let's look in more detail at `std::unique_ptr`. Then I'll tell you about other smart pointers.

Unique Pointers in More Detail

You have seen how to create a `std::unique_ptr` using `make_unique`, and you've used operator `->` to call the member functions of the *pointee* (the thing a pointer points to). What makes this pointer unique? You cannot copy a `std::unique_ptr`. If you try, you'll get a compiler error. That's all (and it's very useful).

Try this:

```

auto asset{ std::make_unique<Stock>("Coffee", 4.8, 0.0113) };
auto try_to_copy{asset};

```

The error might be verbose. For example, here's what Visual Studio reports:

```

error C2280: 'std::unique_ptr<stock_prices::Stock, ❶
    std::default_delete<stock_prices::Stock>
    >::unique_ptr( ❷
        const std::unique_ptr<stock_prices::Stock, ❸
            std::default_delete<stock_prices::Stock>> &)' ❹
    : attempting to reference a deleted function
see declaration of 'std::unique_ptr<stock_prices::Stock,
    std::default_delete<stock_prices::Stock>
    >::unique_ptr'
'std::unique_ptr<stock_prices::Stock,
    std::default_delete<stock_prices::Stock>
    >::unique_ptr(const std::unique_ptr<stock_prices::Stock,
        std::default_delete<stock_prices::Stock>> &)'
    : function was explicitly deleted ❺

```

- ❶ States an error using `std::unique_ptr`
- ❷ ...referring to a `unique_ptr` function
- ❸ ...taking a `const` reference to something
- ❹ States a function is deleted

Buried in the message, you can see that a function has been deleted—specifically, one called `std::unique_ptr`. This is a constructor, which takes a `const` reference, in other words, the copy constructor.

You know how to delete copy constructors—you did it in “Copying Objects” on page 221. A `std::unique_ptr` cannot be copied, either, because its copy constructor has been deleted. This means that only one variable can own the pointee, which makes the implementation lightweight. The unique pointer deletes the pointee when it goes out of scope, because it doesn’t have to worry about any other object potentially having a copy.

If you have more than one object pointing to the same place, when does the delete need to happen? “Other Smart Pointers” on page 243 will explain how C++ approaches shared pointers.

Now, I also claimed that the underlying pointer is deleted when the smart pointer goes out of scope, but that’s not *entirely* true. The `std::default_delete` in the verbose error message hints that you can configure what to do when the pointer goes out of scope. I’ll tell you more in “Custom Deleters” on page 244, but first, let’s finish thinking about what you can do with a `std::unique_ptr`.

You cannot copy the pointer, but you can move it:

```
auto moved_asset{ std::move(asset) }; ❶  
assert(asset == nullptr); ❷
```

- ❶ Moves the asset (pointer) to another `unique_ptr`
- ❷ Asserts that the asset now owns nothing

Notice the `assert` to test what has happened to the pointer moved from `asset`. The original `std::unique_ptr` becomes a `nullptr`, so it no longer owns anything. The unique pointer stays unique.

You can let go of the resource (for example, your `asset`) by calling the `release` function. When you do that, your smart pointer will no longer clean up for you, so you might need to call `delete`. `Release` returns the pointer and, like `move`, sets the `std::unique_ptr` to `nullptr`:

```

auto another_moved_asset{ moved_asset.release() }; ❶
assert(moved_asset == nullptr); ❷
delete another_moved_asset; ❸

```

- ❶ Relinquishes the `moved_asset` (pointer), returning a raw `Stock` pointer
- ❷ Asserts the `moved_asset` now owns nothing
- ❸ Deletes the raw pointer

Other Smart Pointers

The `<memory>` header provides two other smart pointers that are more advanced—you won't need to use them if you're trying the code in this book. One of the two, called a `std::shared_ptr`, can be copied. It can have more than one owner, which adds overhead to track when all sharers are finished with the underlying object. The other smart pointer, called a `std::weak_ptr`, can be used to observe what's in a `std::shared_ptr` but doesn't own it. The `std::shared_ptr` is another class template. You can create one with `std::make_shared`:

```

auto shared_asset{ std::make_shared<Stock>("Coffee", 4.8, 0.0113) };

```

You can then copy this to another `std::shared_ptr<Stock>`:

```

auto joint_asset{ shared_asset };

```

When one asset goes out of scope, it no longer instantly deletes the underlying pointer. The shared pointer counts how many objects are sharing the pointee and calls `delete` when the count drops to zero.

The weak pointer doesn't own anything but can watch a shared pointer:

```

std::weak_ptr just_looking{ joint_asset };

```

You can't do much with a weak pointer. If you want to access the shared pointer that the weak pointer is watching, you call `lock`. This gives you a `std::shared_ptr` back. You'll need to check it for validity, because the owner may have deleted the object:

```

if (auto now_using = just_looking.lock()) ❶
{
    std::cout << "Using the shared pointer\n";
}
else
{
    std::cout << "Unable to use the shared pointer\n";
}

```

- ❶ Tries to get a `std::shared_ptr` from the `std::weak_ptr` and checks it is valid

You could use a `std::weak_ptr` to observe cache data. The cache owns the data, but the observer just watches. If you can't get a `std::shared_ptr` from the lock, that might mean that the data has updated.

Shared and weak pointers are harder to use than a `std::unique_ptr`. You can always start with a `std::unique_ptr` and change your mind later. Start simple.

There's one more important detail about smart pointers to cover: custom deleters.

Custom Deleters

By default, when a unique pointer goes out of scope, its destructor deletes the pointee. However, you can override this behavior. The unique and shared pointers allow you to specify a *custom deleter*: a function to call when the unique pointer goes out of scope or the shared pointer's reference count drops to zero. By default, `delete` is called for you, which calls an object's destructor and releases memory.

You can use a smart pointer to handle other resources than memory, though. More advanced code might use a *socket*: an object to send or receive data over a network or a *handle*: an object using a resource (such as a socket, window, database, or printer connection). Such resources need to be released or closed, and you can use a custom deleter to do this.

You had a value on the stack earlier:

```
int value{ 42 };
```

You can make a `std::unique_ptr` to `value`, but don't let it call `delete`. You only use `delete` for heap memory. Instead, you can provide a custom deleter to change the behavior. There are various ways to do this. We'll do it by defining a struct with a call operator (which you met in “[Binary Operators and Predicates](#)” on page 106). The operator must take a pointer to the type held by the unique pointer. For now, you can use `auto` rather than getting more specific about the type:

```
struct no_op_deleter ❶
{
    void operator()(auto value) const ❷
    {
        std::cout << "Nothing to do for " << *value << "\n"; ❸
    }
};
```

- ❶ Declares a struct
- ❷ Defines a call operator taking `auto`

- ❸ Displays a message, using the dereference operator `*` to get the value from the pointee

I will explain the `auto` parameter further in [Chapter 15](#).

Now you can point to the value on the stack. Instead of using `std::make_unique`, you'll use the address of the value, `&value`, and state the custom deleter's type:

```
std::unique_ptr<
    int, ❶
    no_op_deleter ❷
>
smart_pointer_to_value{ &value }; ❸
```

- ❶ Specifies type of pointee
- ❷ Specifies what to do in the destructor
- ❸ Initializes a smart pointer with the address of `value`

The `std::unique_ptr` now uses your custom deleter instead of the default `delete`. When `smart_pointer_to_value` goes out of scope, its destructor will be called. The destructor calls your custom deleter's operator, so your message is displayed:

```
Nothing to do for 42
```

Using custom deleters is intricate. You can find details on the internet. For example, CppStories shows [how to use custom deleters](#), and Herb Sutter's blog gives lots more detail on [smart pointers](#). You can also avoid using custom deleters by defining a class that tidies up in the destructor, in which case you're using RAII.

Using a `std::unique_ptr` in a Class

Let's round off this section using a `std::unique_ptr` differently to reinforce what you have learned so far.

Make two new files: *trade.cpp* and *trade.h*. You are going to write an `Exchange` class, representing a marketplace where people can buy and sell stocks. In [Chapter 13](#) you will add functionality to `Exchange` that allows you to buy or sell items. In this section, you will get some of the structure into place. I will introduce a couple of new C++ features, and we'll revise some ideas you have already met.

Add the definition of an `Exchange` class inside the `stock_prices` namespace in the header, as shown in [Example 12-1](#).

Example 12-1. Start of an Exchange class

```
#include <memory>
#include <vector>

#include "stock.h"

namespace stock_prices
{
    class Exchange ❶
    {
        double initial_funds{}; ❷
        double funds{ initial_funds }; ❸
        int number_of_assets{}; ❹
        std::unique_ptr<Stock> asset{}; ❺
        std::vector<double> prices{}; ❻

        public:
        Exchange(int number_of_assets)
            : number_of_assets(number_of_assets) ❼
        {
        }
        double next_price(); ❽
        std::vector<double> get_prices() const ❾
        {
            return prices;
        }
    };
}
```

- ❶ Defines an Exchange class in the stock_prices namespace
- ❷ Declares the initial funds
- ❸ Sets the current funds to the value of initial_funds
- ❹ Tracks the number of assets
- ❺ Declares a std::unique_ptr to a Stock
- ❻ Uses a std::vector to track historical prices
- ❼ Defines a constructor taking the number of assets
- ❽ Declares a next_price function, which you will define shortly
- ❾ Defines a function to get historical prices

Don't forget, the data members are `private`, because a class's members are `private` by default. The constructor and price functions come after the public access modifier, so they can be used from outside the class.

The Exchange is based on the original trading game you wrote in [Example 7-2](#), but this one won't play fair. For example, the Exchange can have negative funds.

You have declared a `next_price` function, but you haven't defined it. Open your `trade.cpp` file and add the definition:

```
#include <stdexcept>

#include "trade.h"

double stock_prices::Exchange::next_price() ❶
{
    if (!asset) ❷
    {
        throw std::invalid_argument{"No asset available"}; ❸
    }
    prices.push_back(asset->get_price()); ❹
    return const auto price = asset->next_price(); ❺
}
```

- ❶ Defines the `next_price` function
- ❷ Checks that the `std::unique_ptr` owns something
- ❸ Throws an exception if there is no asset yet
- ❹ Stores the current price
- ❺ Returns the asset's next price

Now, you haven't got a way to set the `asset` yet—we'll come back to that. First, let's talk about the constructor taking one argument. You can create an exchange in a `main.cpp` file and try to show prices, as shown in [Example 12-2](#). (You don't have any prices yet, but you will learn something.)

Example 12-2. Spot the deliberate mistake

```
#include <iostream>

#include "trade.h"

void show_prices(const stock_prices::Exchange & exchange) ❶
{
    for(auto price: exchange.get_prices())
```



```

    {
        std::cout << price << '\n';
    }
}

int main()
{
    using namespace stock_prices;
    Exchange exchange{100}; ❷
    show_prices(100); ❸
}

```

- ❶ Takes an Exchange to display prices
- ❷ Creates an Exchange
- ❸ Forgets to use the Exchange and uses a number instead

The code compiles, even though you passed 100 to the `show_prices` function instead of an `Exchange`. An `Exchange` can be constructed from an `int`, so passing 100 creates a temporary `Exchange` for use in the function. That is a bad idea. You can avoid this by marking the constructor as `explicit`:

```

explicit Exchange(int number_of_assets) ❶
    : number_of_assets(number_of_assets)
{
}

```

- ❶ Stops you from being able to construct an `Exchange` implicitly

If you add the single keyword `explicit`, the `main` function can no longer send 100 to the `show_prices` function.



Always mark a single parameter constructor as `explicit`, unless you want to be able to implicitly create your class from an instance of the parameter's type. An implicit conversion can be useful, but it might create a temporary from the parameter type, or you may find an overload called that you weren't expecting.

Let's add another constructor that takes a count of assets, along with a smart pointer to a `Stock`, as shown in [Example 12-3](#).

Example 12-3. An improved `Exchange` class

```

class Exchange
{
    double initial_funds{};
}

```

```

double funds{ initial_funds };
int number_of_assets{};
std::unique_ptr<Stock> asset{};
std::vector<double> prices{};

public:
    Exchange(int number_of_assets, std::unique_ptr<Stock> asset) ❶
        : number_of_assets(number_of_assets), asset(std::move(asset)) ❷
    {
    }
    explicit Exchange(int number_of_assets) ❸
        : Exchange(number_of_assets, nullptr) ❹
    {
    }
    double next_price();
    std::vector<double> get_prices() const
    {
        return prices;
    }
};

```

- ❶ Takes a `std::unique_ptr` of `Stock` by value
- ❷ Moves the `asset` to the member variable
- ❸ Defines an explicit constructor
- ❹ Delegates to the other constructor, using a `nullptr` as the `asset`

The single parameter constructor is now explicit and *delegates* (passes responsibility) to the two-parameter constructor. A *delegating constructor* uses the class name in the member initializer list, passing any relevant parameters. The code listing passes `number_of_assets` and `nullptr` to the constructor, taking two arguments. You can now make an `Exchange` with or without an `asset`.

Using the Exchange Class

Let's add a test function to revise it and learn about the details of your new class. Add a declaration for the function to the *trade.h* file inside the namespace:

```
void test_trades();
```

Add the definition inside the *trade.cpp* file and call the function from `main`.

Let's start with an exchange with no assets. The exchange should throw an exception when you ask for the price:

```

#include <cassert>

#include "trade.h"

void stock_prices::test_trades()
{
    using namespace stock_prices;
    Exchange exchange{ 100 }; ❶
    try ❷
    {
        exchange.next_price(); ❸
        assert(false); ❹
    }
    catch (const std::exception & ) ❺
    {
    }
}

```

- ❶ Creates an Exchange with no asset
- ❷ Tries to get the next price
- ❸ Asserts false, since you should not get to this line
- ❹ Catches an exception

You wrote similar tests back in “Starting with a Failing Test” on page 24. Let’s add some more tests.

Now, the Exchange contains a `std::unique_ptr`. You know you can’t copy a unique pointer. This means you can’t copy an Exchange, either. The following code will not compile:

```

Exchange exchange{ 100 };
Exchange copied_exchange(exchange);

```

The error message varies between compilers, but in essence, it says that the copy constructor is implicitly deleted because a member’s copy constructor is deleted.

Testing code that won’t compile is a small challenge. C++ allows you to check whether the special member functions you met in Chapter 11 are available. The `<type_traits>` header provides various ways to check the *properties*, or traits, of types. You can ask if a class has a copy constructor or copy assignment using this:

```

static_assert(std::is_copy_constructible<Exchange>()); ❶
static_assert(std::is_copy_assignable<Exchange>()); ❷

```

- ❶ Checks if Exchange has a copy constructor

❷ Checks if Exchange has a copy assignment operator

Notice that I've used `static_assert` rather than `assert`. The type traits provide *metaprogramming* facilities: a way to program at compile time, including testing with `static_assert`. So you can use `static_assert` for compile-time checks. However, this limits what you can check. The type traits work because they involve the type of an object, rather than values that are known only at runtime. If a `static_assert` fails, the program will not compile. Failing to compile is better than strange behavior at runtime. I think being able to test your code at compile time is amazing.

Try adding these two checks to your `test_trades` function. You should now get a compile error saying that the static assertions failed.

Let's make the tests pass, adding a `!` (for “not”) to the checks:

```
static_assert(!std::is_copy_constructible<Exchange>()); ❶  
static_assert(!std::is_copy_assignable<Exchange>()); ❷
```

❶ Checks Exchange does not have a copy constructor

❷ Checks Exchange does not have a copy assignment operator



You don't need to put a `static_assert` inside a function. You can put it inside your namespace or any block as a standalone statement. I have grouped these with the runtime asserts, so they are all in one place.

You created a class containing a `std::unique_ptr`. Your class can't copy this asset, so your class can't be copied. You can check *at compile time* that this happens.

For completeness, add a test creating an `Exchange` with an asset:

```
Exchange exchange_with_asset{  
    100, std::make_unique<Stock>("Coffee", 4.8, 0.0113)  
}; ❶  
assert(exchange_with_asset.get_prices().empty()); ❷  
exchange_with_asset.next_price(); ❸  
assert(exchange_with_asset.get_prices().size() == 1); ❹
```

❶ Creates an Exchange with an asset

❷ Checks that you start with no historical prices

❸ Gets (and discards) the next price

❹ Checks that you then have one price

You will use the `Exchange` class again in the next chapter but will make a few tweaks once you have learned more.

Conclusion

You used `std::make_unique` to create a `std::unique_ptr` of `Stock` on the heap. You learned how to use operator `->` to call the `Stock`'s member functions. A `std::unique_ptr` cannot be copied because its copy constructor is deleted, but it can be moved. You saw that a class containing a `std::unique_ptr` cannot be copied, either. Unique pointers enforce unique ownership, and they should be your first port of call when you want a smart pointer. You'll use unique pointers in [Chapter 13](#).

You learned a little about the other smart pointers, `std::shared_ptr` and `std::weak_ptr`, as well as custom deleters. The other smart pointers are for more complicated scenarios and are more heavyweight—they use more memory because they have more to do. The shared pointer has a reference count to track how many pointers are watching a pointee. When the count drops to zero, the deleter is called. A weak pointer therefore needs to check if it can get the underlying shared pointer from a call to `lock`, in case the pointee has been deleted.

You also met RAII, “Resource Acquisition Is Initialization,” one of the most important idioms of C++. This means acquiring resources, usually in a constructor, and releasing them in the destructor. For smart pointers, the resource is often heap memory, but RAII applies to any situation where a resource needs releasing.

You practiced using raw pointers and references, too. You used a raw pointer to stack memory, specifically an `int`. Practicing with simple types is often helpful. You also learned that a reference always refers to the same object, but a pointer can be switched to point elsewhere. A pointer might also be `nullptr`, or point to memory that has been deleted. Raw pointers are hard to work with.

You met `explicit` constructors, which stop you from accidentally making a temporary object via a single-parameter constructor. You also used `static_assert` to test type traits at compile time.

You've covered a lot of ground now. There are a few more C++ features to cover, and then you will know enough to be able to build a variety of programs. You have seen more than one way to generate stock prices. So far, your `Stock` class uses one `next_price` method. You can vary member functions' behavior using the keyword `virtual`. The next chapter explains what this means and how to build a class hierarchy so you can vary behavior.

Classes: Virtual Functions and Inheritance

You met classes in [Chapter 10](#), and did a deep dive into special member functions in [Chapter 11](#). You also learned how to use smart pointers in [Chapter 12](#). In this chapter, you will use these building blocks to build a better trading game.

You can make a *hierarchy* of classes: a way to relate classes from abstract general concepts to more specific types. You can then vary behavior in related classes. Your `Stock` uses one method to get a price, but you have seen other approaches. This chapter will show you how to provide related classes that use different strategies to price an asset. Using different classes related by a hierarchy to vary behavior—a family of types—is called *object-oriented programming* (OOP).

If you work on an existing codebase with a hierarchy of classes, you can easily add a new related class to change behavior, for example, add a new feature to an app. Without the related classes, you could have large functions with lots of `ifs/elses` to vary behavior. Adding a new feature might involve adding even more `ifs` and `elses`, leaving you with massive functions that are hard to reason about. OOP gives you a (relatively) simple way to add new features.

I will show you how to make a specific `Stock` from a general `Asset` class, so you can use it in a new trading game. You will then make another class in [“Adding Another Derived Type” on page 264](#) and be able to vary how you play the game, without changing the game code itself.

You will also get the chance to revise C++ features you already know, including `static_assert` and exceptions.

Base Class and Derived Classes

A hierarchy arranges related classes into levels. The best arrangement uses an *abstract* class as a base for the others. The class is abstract when it declares some functions but does not implement them. The abstract class provides an *interface*: a description of what you can do with a related class.

You can have a reference or pointer to an abstract class but cannot create an instance. Other classes can be *derived* from the base class, so you refer to or point to a derived class instead. A derived class has its own implementation of the base's unimplemented functions, allowing behavior to vary. This is useful, because you can then program to an interface (the base class), without worrying about details, and add new derived classes to vary behavior without changing the code using the interface.

Defining an Abstract Base Class

Let's start with a base class and see how to derive other classes from it later. You indicate a member function can be reimplemented in a different way in derived classes using the keyword `virtual`. You can also add `= 0` in a base class, rather than provide an implementation. This member function is called a *pure virtual function*: one with no implementation. The base class is then *abstract*; you can't make an instance of it.

Create a new file called *asset.h*. The original `Stock` has a `get_name` and `next_price` function. For a more general asset, you will want both of these functions. It might be useful to have another that returns the current price. Add a new class called `Asset` with three abstract virtual functions, inside the `stock_prices` namespace. You also need to provide a virtual destructor, for reasons I will explain in “[Virtual Functions and Inheritance in Depth](#)” on page 271:

```
#pragma once

#include <string>

namespace stock_prices
{
    class Asset
    {
    public:
        virtual ~Asset() = default; ❶
        virtual std::string get_name() const = 0; ❷
        virtual double get_price() const = 0; ❸
        virtual double next_price() = 0; ❹
    };
}
```

- ❶ Provides a default virtual destructor
- ❷ Declares a pure `get_name` const virtual function
- ❸ Declares a pure `price` const virtual function
- ❹ Declares a pure `next_price` virtual function

Let's make a derived class, implementing the pure virtual functions.

A Derived Class

Your existing `Stock` class nearly does everything an `Asset` does. Find your *stock.h* file so you can make the required changes.

A derived class needs to state which base it is implementing. Include *asset.h* so you can use your abstract `Asset` class. You add a colon after your class name, then `public`, followed by the name of the base class:

```
class Stock: public Asset
```

You can have more than one base class, but having only one keeps things simple. You met `public`, `protected`, and `private` as access modifiers in [Chapter 10](#). `Protected` and `private` inheritance are for niche use cases, beyond beginner level.

So, that's the first line of your `Stock` class. You already have a `get_name` and `next_price` function, but not the new `get_price` function, so `Stock` is currently abstract. Recall, that means at least one virtual function is still pure. Trying to create a `Stock` causes a compile error. Try this, somewhere in your `main` function:

```
auto coffee{ Stock{ "Coffee", 4.8, 0.0113 } };
```

Don't forget, you need to use the *stock.cpp* file as well as *main.cpp* in your build, and include *stock.h*. You will get an error saying something like:

```
'stock_prices::Stock': cannot instantiate abstract class
```

To create a `Stock` you must provide a `get_price` function. The implementation doesn't need much, so can go inline in the class definition in the *stock.h* header file:

```
double get_price() const
{
    return last_price;
}
```

The functions are automatically `virtual` since the `Asset` base class provides a `virtual` function with exactly the same signature. But, as a safety precaution, you can mark them as `override`. If you get the function signature wrong in any way, for

example, forgetting the `const`, the compiler will tell you if you used `override`. Without the `override`, a wrong signature means you have written a different function, and you won't get a warning or error.

Your improved stock header looks like this:

```
#pragma once

#include <random>
#include <string>

#include "asset.h" ❶

namespace stock_prices
{
    class Stock: public Asset ❷
    {
        std::string name{};
        double last_price{};
        double volatility{};
        std::default_random_engine gen;
        std::normal_distribution<double> distrib;

    public:
        Stock(const std::string & stock_name,
              double start_price,
              double start_volatility);

        Stock(const Stock & other) = delete;
        Stock(Stock && other) noexcept = default;

        Stock & operator = (const Stock & other) = delete;
        Stock & operator = (Stock && other) noexcept = default;

        std::string get_name() const override ❸
        {
            return name;
        }

        double get_price() const override ❹
        {
            return last_price;
        }

        double next_price() override; ❺
    };
}
```

- ❶ Includes the asset header
- ❷ Declares public inheritance from Asset

- ❸ Overrides the pure virtual `get_name` function
- ❹ Overrides the pure virtual `get_price` function
- ❺ Overrides the pure virtual `next_price` function (which is implemented in the `.cpp` file)

You don't need to explicitly provide the destructor, because the compiler will provide one for you. You can now make a `Stock` class as you did before in [Chapter 10](#):

```
stock_prices::Stock coffee{"Coffee", 4.8, 11.3};
```

Now `Stock` derives from an abstract base class and provides implementations for all the pure virtual functions. The `Stock` is therefore called a *concrete* class: a class you can create an instance of. The concrete class extends the base class, so the derived class contains the base as a subobject.

When you construct the concrete class, it constructs the base first and then its own members. Destruction happens in reverse order. [Figure 13-1](#) illustrates how the classes are related and how construction and destruction work.

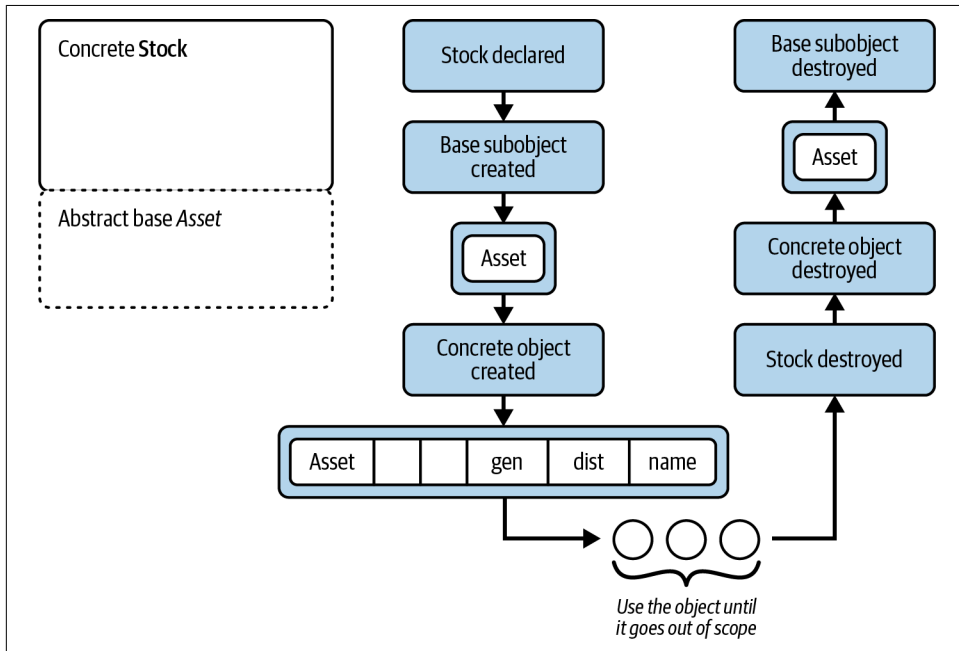


Figure 13-1. A concrete class deriving from an abstract class

The concrete `Stock` class is an `Asset` so behaves as though it contains an `asset` as well as its own members. The C++ standard does not guarantee any layout or position for

the base class, and you sometimes get spaces (called *padding*) between members. Think of a base and derived class as stacking up the members: the base is constructed first and then the derived. Destruction happens in reverse order.

I'll show you how to use the `Stock` class via the interface. Afterward, you will create another type of asset to see the power and usefulness of derived classes.

Using Derived Classes

You have already used and written overloaded functions to provide different behavior depending on an object's type. This is one kind of *polymorphism*: many (poly) changes or forms (morphs), called *static polymorphism*: the compiler finds the right version of a function based on the types of the parameters, so this happens at compile time. If you use derived classes, you can vary the type at runtime, which is called *dynamic polymorphism*.

You can have tests for your new types, using the type traits you met in “Using the Exchange Class” on page 249. Add a new `test_stock` function like this:

```
#include <type_traits> ❶

#include "stock.h"

void stock_prices::test_stock() ❷
{
    using namespace stock_prices;
    static_assert(std::is_abstract<Asset>()); ❸
    static_assert(std::is_polymorphic<Asset>()); ❹
    static_assert(std::is_polymorphic<Stock>()); ❺
}
```

- ❶ Includes the `type_traits` header as before
- ❷ Defines the test function
- ❸ Asserts the base class is abstract
- ❹ Asserts the base class is polymorphic
- ❺ Asserts the derived class is polymorphic

Remember the `static_assert` is checked at compile time. Static means compile time; dynamic means runtime.

Let's learn how to use the `Stock` class via the base class. You cannot make an instance of an abstract base class. You can use a reference or pointer to one though, provided you refer or point to a concrete class. Try using both a reference and a smart pointer:

```

#include <iostream>
#include <memory>

#include "asset.h"
#include "stock.h"

int main()
{
    using namespace stock_prices;
    auto coffee{ Stock{ "Coffee", 4.8, 0.0113 } }; ❶
    Asset & asset{coffee}; ❷
    std::cout << asset.get_name() << ": " << asset.next_price() << '\n'; ❸

    std::unique_ptr<Asset> asset_pointer{
        std::make_unique<Stock>("Coffee", 4.8, 0.0113)
    }; ❹
    std::cout << asset_pointer->get_name() << ": "
        << asset_pointer->next_price() << '\n'; ❺
}

```

- ❶ Creates an instance of the concrete type `Stock`
- ❷ Refers to the `Stock` via the abstract base class
- ❸ Uses the base class's interface
- ❹ Creates a unique pointer to another `Stock`
- ❺ Also uses the base class's interface via the smart pointer

Notice both `asset` and `asset_pointer` are using the abstract `Asset`. The `Asset` provides an interface, which you use for various different concrete classes. Relying on references can be difficult. You have to ensure the object they alias remains in scope while you use the reference. That's simple enough in a short `main` function, but for a larger application, tracking lifetimes is harder. A smart pointer is therefore the preferred approach for OOP.

You created an `Exchange` class in [Example 12-3](#), which uses a `std::unique_ptr` to a `Stock` class. You can change that to the abstract `Asset` now. In this section, you will still use a `Stock`, but you can extend your code to use other types of assets without needing to change the `Exchange` class further. Your `Exchange` will use the abstract interface and does not need to know implementation details. Make two changes to use the abstract `Asset`, and include the `asset.h` header, as shown in [Example 13-1](#).

Example 13-1. An further improved Exchange class using the abstract Asset

```
#pragma once

#include <memory>
#include <vector>

#include "asset.h" ❶

namespace stock_prices
{
    class Exchange
    {
    public:
        double initial_funds{100.0};
        double funds{ initial_funds };
        int number_of_assets{};
        std::unique_ptr<Asset> asset{}; ❷
        std::vector<double> prices{};

        Exchange(int number_of_assets, std::unique_ptr<Asset> asset) ❸
            : number_of_assets(number_of_assets), asset(std::move(asset))
        {
        }
        explicit Exchange(int number_of_assets)
            : Exchange(number_of_assets, nullptr)
        {
        }
        double next_price();
        std::vector<double> get_prices() const
        {
            return prices;
        }
    };

    double trading_game(Exchange & exchange);
}
```

- ❶ Includes the asset header (instead of the stock header)
- ❷ Defines a data member of a `std::unique_ptr` to an `Asset`
- ❸ Declares a constructor taking a `std::unique_ptr` of `Asset` by value

You can create an `Exchange` in `main.cpp`. Call your test functions too:

```
#include "asset.h"
#include "stock.h"
#include "trade.h"

int main()
{
}
```

```

using namespace stock_prices;
test_stock(); ❶
test_trades(); ❷

std::unique_ptr<Asset> asset{
    std::make_unique<Stock>("Coffee", 4.8, 0.0113) ❸
};
Exchange exchange{1, std::move(asset)}; ❹
}

```

- ❶ Calls the stock tests
- ❷ Calls the trades tests
- ❸ Create a `std::unique_ptr` to coffee Stock
- ❹ Moves the asset to an Exchange

Let's use the Exchange in a new trading game.

A New Trading Game

Start by adding a declaration of the game to your *trade.h* file:

```
double trading_game(Exchange & exchange);
```

You pass the exchange by reference, because the prices will change during the game. The function returns a profit. You will implement the game in *trade.cpp*.

The game itself will be familiar, but you use an Exchange rather than the stock directly. Let's add functions to Exchange to buy or sell stock. To keep things simple, let the number of assets or funds go negative:

```

void fulfill_buy_order()
{
    --number_of_assets;
    funds += asset->get_price();
}

void fulfill_sell_order()
{
    ++number_of_assets;
    funds -= asset->get_price();
}

```

When a player buys, the Exchange hands over an asset and takes the money. When a player sells, the Exchange takes the asset and spends money. In real life, the Exchange might charge a fee or offer different prices for buying or selling—and might not be allowed to sell assets they don't own! They may not be allowed to go too far

overdrawn either. But this is a simulation to help you learn C++, so that's OK. You can add extra checks yourself for practice if you want.

Let's write a trading game using the `Exchange`, based on [Example 9-1](#). Instead of looping over the prices, you now allow a player to quit by typing 'q', as shown in [Example 13-2](#).

Example 13-2. A new trading game using the `Exchange` class

```
#include <cassert>
#include <format>
#include <iostream>
#include <print>

#include "trade.h"

double stock_prices::trading_game(Exchange & exchange)
{
    const double initial_funds{ 100.0 };
    double funds{ initial_funds };
    int number_of_shares{};

    bool playing{ true }; ❶
    while(playing) ❷
    {
        auto status = std::format("Funds ${:.2f}, Shares {}",
                                   funds, number_of_shares);
        std::println!("{}", status);
        const auto price = exchange.next_price(); ❸
        auto price_message = std::format("Current price: ${:.2f}", price);
        std::println("[: >{}}", price_message, status.size());
        std::println("Press (s) to sell, (b) to buy, (q) to quit");
        std::print("or something else to continue>");
        char choice{};
        std::cin >> choice;

        if (choice == 's')
        {
            if (number_of_shares > 0)
            {
                exchange.fulfill_sell_order(); ❹
                --number_of_shares;
                funds += price;
            }
            else
            {
                std::println("No stock to sell");
            }
        }
        else if (choice == 'b')
        {

```

```

        if (price <= funds)
        {
            exchange.fulfill_buy_order(); ❸
            ++number_of_shares;
            funds -= price;
        }
        else
        {
            std::println("Insufficient funds");
        }
    }
    else if (choice == 'q') ❹
    {
        playing = false;
    }
}
return funds - initial_funds;
}

```

- ❶ Tracks if the user still wants to play
- ❷ Loops while the game is ongoing
- ❸ Gets the next price from the exchange
- ❹ Sells an asset to the exchange
- ❺ Buys an asset from the exchange
- ❻ Allows the player to quit

Call your game from main:

```

#include <iostream>
#include <memory>

#include "asset.h"
#include "stock.h"
#include "trade.h"

int main(int argc, char *argv[])
{
    using namespace stock_prices;
    test_stock(); ❶
    test_trades(); ❷

    std::unique_ptr<Asset> asset{
        std::make_unique<Stock>("Coffee", 4.8, 0.0113) ❸
    };
    Exchange exchange{1, std::move(asset)}; ❹
}

```



```

    auto profit = trading_game(exchange); ❸
    std::cout << "Total profit " << profit << '\n'; ❹
    std::cout << "Game over\n";
}

```

- ❶ Runs stock tests
- ❷ Runs trades tests
- ❸ Creates an asset
- ❹ Creates an exchange
- ❺ Runs the game
- ❻ Reports the profit

Build and run your game. The game shows your funds and shares, offering you options to buy, sell, or quit, and continues until you quit (any other letter continues the game):

```

Funds $100.00, Shares 0
  Current price: $4.85
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>b
Funds $95.15, Shares 1
  Current price: $4.86
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>s
Funds $100.01, Shares 0
  Current price: $4.88
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>q
Total profit $0.01
Game over

```

Now, the exchange keeps track of prices as you play the game. You can replay the game using the saved prices if you make a different Asset that uses a vector of prices.

Adding Another Derived Type

Let's make a new Asset type that uses a vector of historical prices. Start the new class in a new *historical_prices.h* file.

Again, derive your new class publicly from Asset. You can probably fill in some of the details without my help, overriding the `get_name` function, and declaring the price functions:

```

#include <string>
#include <vector>

#include "asset.h"

namespace stock_prices
{
    class HistoricalPrices : public Asset ❶
    {
        std::string name{};
        std::vector<double> prices{};
        size_t index{0}; ❷

    public:
        explicit HistoricalPrices(const std::vector<double> & prices) ❸
            : prices(prices)
        {
        }
        std::string get_name() const override ❹
        {
            return name;
        }

        double get_price() const override;

        double next_price() override;
    };
}

```

- ❶ Says HistoricalPrices is a new type derived from Asset
- ❷ Stores an index into the prices
- ❸ Marks one-parameter constructor as explicit
- ❹ Overrides the base's pure virtual functions

Now you need to implement the virtual price function, in a new *historical_prices.cpp* file. `next_price` can use `get_price`, incrementing the index on each call. `get_price` can return the price at the current index, under one condition.

Did you spot the potential edge case? You need a way to indicate when you get to the end of the prices. If you thought of this, well done. You could throw a `std::exception` when you run out of prices, but you have seen more specific exception types, like the `std::invalid_argument` you met in “Other Exception Types” on page 52. If you throw a more specific type, your trading game can catch this and end the game.

The `std::exception` has a virtual destructor, so you can use it as a base class. The simplest thing to do is to derive publicly from `std::exception`. Add this to your *asset.h* file:

```
#pragma once

#include <exception> ❶
#include <string>

namespace stock_prices
{
    class no_more_prices : public std::exception ❷
    {
    };
}
```

❶ Includes the `<exception>` header

❷ Defines a new exception type

Now you can implement your prices functions, letting `get_price` throw the new exception if the prices run out, as shown in [Example 13-3](#).

Example 13-3. Implementation of virtual price function

```
#include "historical_prices.h"

double stock_prices::HistoricalPrices::next_price()
{
    ++index; ❶
    return get_price(); ❷
}

double stock_prices::HistoricalPrices::get_price() const
{
    if (index == prices.size()) ❸
    {
        throw no_more_prices{}; ❹
    }
    return prices[index]; ❺
}
```

❶ Increments the `index`

❷ Returns whatever `get_price` returns

❸ Checks `index` hasn't run over the end of the prices

- ❹ Throws your exception if there are no prices left
- ❺ Returns a price if there are some left

Let's use your new class at the end of the game so the player can retry their strategy against the historical prices, as shown in [Example 13-4](#).



In finance, people will try trading strategies against various historical prices and call this backtesting. Doing so can give a hint of how good the strategy is, but past performance is not indicative of future results (as people often point out).

Example 13-4. A new trading game catching exceptions

```
double stock_prices::trading_game(Exchange & exchange)
{
    const double initial_funds{ 100.0 };
    double funds{ initial_funds };
    int number_of_shares{};

    bool playing{ true };
    while(playing)
    {
        auto status = std::format("Funds ${:.2f}, Shares {}",
                                   funds, number_of_shares);
        std::println!("{}", status);
        try ❶
        {
            const auto price = exchange.next_price();
            auto price_message = std::format("Current price: ${:.2f}", price);
            std::println("{: >{}}", price_message, status.size());
            std::println("Press (s) to sell, (b) to buy, (q) to quit");
            std::print("or something else to continue>");
            char choice{};
            std::cin >> choice;

            if (choice == 's')
            {
                if (number_of_shares > 0)
                {
                    exchange.fulfill_sell_order();
                    --number_of_shares;
                    funds += price;
                }
                else
                {
                    std::println("No stock to sell");
                }
            }
        }
    }
}
```

```

        else if (choice == 'b')
        {
            if (price <= funds)
            {
                exchange.fulfill_buy_order();
                ++number_of_shares;
                funds -= price;
            }
            else
            {
                std::println("Insufficient funds");
            }
        }
        else if (choice == 'q')
        {
            playing = false;
        }
    }
    catch(const no_more_prices &) ❷
    {
        break; ❸
    }
}
return funds - initial_funds;
}

```

- ❶ Wraps the call to `next_price` in a try block
- ❷ Catches the specific exception for no more prices
- ❸ Breaks out of the loop, stopping the game

Include the *historical_prices.h* file in *main.cpp*, and add an option to rerun the trading game at the end of `main`, as shown in [Example 13-5](#).

Example 13-5. An improved trading game, allowing you to rerun against historical prices

```

#include <iostream>
#include <memory>

#include "asset.h"
#include "historical_prices.h" ❶
#include "stock.h"
#include "trade.h"

int main(int argc, char *argv[])
{
    using namespace stock_prices;
    test_trades();
}

```

```

std::unique_ptr<Asset> asset{
    std::make_unique<Stock>("Coffee", 4.8, 0.0113)
};
Exchange exchange{1, std::move(asset)};
auto profit = trading_game(exchange);
std::cout << "Total profit " << profit << '\n';
std::cout << "Game over\n";
std::cout << "Rerun? [y]es n[o]? \n"; ❷
char choice; ❷
std::cin >> choice; ❷

if (choice == 'y')
{
    Exchange historical_exchange{ 1,
        std::make_unique<HistoricalPrices>(exchange.get_prices()) ❸
    };
    profit = trading_game(historical_exchange); ❹
    std::cout << "Total profit " << profit << '\n';
    std::cout << "Game over\n";
}
}

```

- ❶ Includes header for the new class
- ❷ Asks if the player wants to backtest their strategy
- ❸ Creates an Exchange using historical prices
- ❹ Replays the game

Don't forget to use *historical_prices.cpp* in your build.

You can now retry the game, based on historical prices:

```

Funds $100.00, Shares 0
  Current price: $4.86
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>b ❶
Funds $95.14, Shares 1
  Current price: $4.83
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>f ❷
Funds $95.14, Shares 1
  Current price: $4.91
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>s ❸
Funds $100.05, Shares 0
  Current price: $5.02
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>f

```

```

Funds $100.05, Shares 0
  Current price: $4.89
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>q ❹
Total profit $0.05 ❺
Game over ❺
Rerun? [y]es n[o]?

```

- ❶ Buys at the first tick
- ❷ Selects f (anything other than b, s, or q) to go to the next price
- ❸ Sells because the price went up a bit
- ❹ Quits the game
- ❺ Profit shown and game ends

If you select rerun, you can cheat and sell at the highest price:

```

Funds $100.00, Shares 0
  Current price: $4.86
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>f
Funds $100.00, Shares 0
  Current price: $4.83
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>b ❶
Funds $95.17, Shares 1
  Current price: $4.91
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>f
Funds $95.17, Shares 1
  Current price: $5.02
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>s ❷
Funds $100.19, Shares 0

```

- ❶ Buys on the second tick
- ❷ Sells after the price has gone up twice rather than immediately

Seems like cheating, right? You could try always selling on a second price increase for random prices and see what profit you can make. Play around with different strategies. On average, you are not likely to make any money, but you can relax and play your game for a bit.

Now you can use two different types of asset in your exchange and don't need to change the trading game code. You could extend your code to take a filename as an

argument to `main` and use real prices saved in a file. In [Example 8-3](#), you read prices from a file and displayed them. Your `get_prices` overloaded function returned a `std::vector<double>`, so you know how to do this on your own. You can find various datasets on the internet. Search for commodity prices to find plausible coffee or tea datasets. Try that as an extension for extra practice. Let's look at some of the inner workings of virtual functions, to give you more insight into how OOP works in C++.

Virtual Functions and Inheritance in Depth

I told you that you need a virtual destructor in a polymorphic base class but haven't explained why yet. In C++, a virtual destructor makes it possible to automatically call derived destructors via pointers or references to a base, but a nonvirtual one does not. I will explain why this matters and explain some of the details behind how virtual functions work in C++.

Virtual Destructors

First, let's see both destructors called. Here's code with a `Base` and `Derived` class, doing the right thing:

```
#include <iostream>
#include <memory>

class Base
{
public:
    virtual ~Base()
    {
        std::cout << "\t~Base\n"; ❶
    }
};

class Derived: public Base
{
    std::string name;
public:
    explicit Derived(std::string name)
        : name(std::move(name))
    {

    }

    ~Derived()
    {
        std::cout << name << " ~Derived\n"; ❷
    }
};

int main()
```



```

{
    Derived derived{"stack"}; ❸
    std::unique_ptr<Base> pointer{std::make_unique<Derived>("heap")}; ❹
}

```

- ❶ Logs in the Base virtual destructor
- ❷ Logs in the Derived destructor, which is virtual because the base class is
- ❸ Makes a Derived on the stack
- ❹ Creates a Derived on the heap, stored in a unique_ptr to Base

If you run this, you see both destructors called:

```

heap ~Derived
    ~Base
stack ~Derived
    ~Base

```

In [Figure 13-1](#), you saw the destructors called in reverse order of the constructors. The stack is also unwound in reverse order too: the heap object was made second but is destroyed first. Both objects invoke two destructors.

Now try a bad base class instead, with no virtual destructor:

```

class BadBase
{
public:
    ~BadBase() ❶
    {
        std::cout << "\t~BadBase\n";
    }
};

class Derived: public BadBase
{
    std::string name;
public:
    explicit Derived(std::string name)
        : name(std::move(name))
    {

    }

    ~Derived()
    {
        std::cout << name << " ~Derived\n";
    }
};

```

❶ Defines a nonvirtual destructor

This time you see three destructors called:

```
~BadBase
stack ~Derived
~BadBase
```

When you use a `Derived` type, both destructors are called. When you have a pointer (smart or otherwise) to the base class, a nonvirtual destructor means only the destructor for the pointee is called. That's bad for a couple of reasons. First, a destructor might have some work to do, for example, releasing a resource, so you will have a resource leak. Second, deleting an object through a pointer to the base is undefined behavior **when the destructor is not virtual**. So, remember a destructor in a polymorphic class must be `virtual`.

Virtual Functions and Slicing

Each class in a hierarchy has its own versions of the virtual functions. C++ does not specify how virtual functions should be implemented, but a lookup table of function pointers is often used. This table is called a *vtable* or virtual function table. The specific override called is looked up in the table. Now, you don't need to know the implementation details, but having a mental model can help you reason through other situations.

Think of virtual functions this way: a virtual function's implementation can vary. C++ uses the runtime type of a class to discover which implementation to call. **Figure 13-2** illustrates related classes using a vtable to store the virtual function implementations. The abstract `Asset` has a pointer to a vtable, `vp_ptr`. The concrete classes set these to the appropriate vtable.

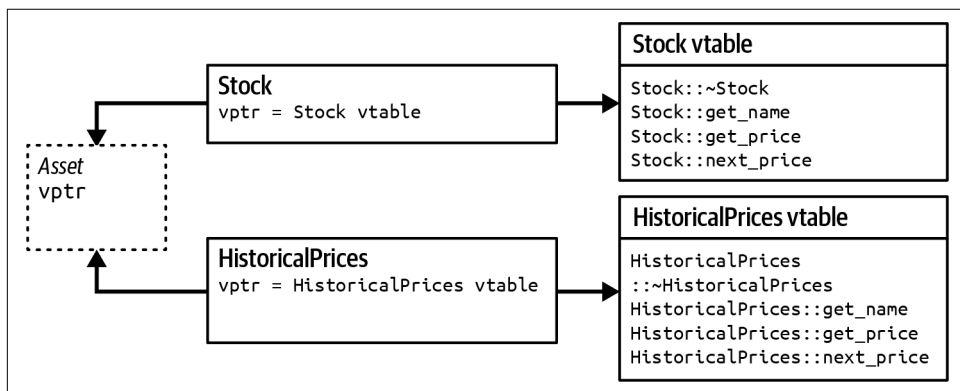


Figure 13-2. Virtual function stored as vtables

Code using the asset doesn't need changing if you add a new derived class, so you can add new features to your code relatively easily. However, some people avoid class hierarchies because calling a virtual function can mean finding which function to call before actually calling it. This extra step (or level of indirection) can slow things down. Under some circumstances, the compiler can *optimize*: leaving out the extra step and calling the appropriate function directly. In [Chapter 14](#) I will show you another way to vary behavior, without using a class hierarchy. However, this alternative approach causes a different set of issues. There are always trade-offs.

Now, the virtualness has implications. You saw how forgetting `virtual` in a destructor is a problem in [“Virtual Destructors” on page 271](#). Returning to the Base and Derived examples will illustrate another problem. What does this code do?

```
#include <iostream>

class Base
{
public:
    virtual void show_number() const
    {
        std::cout << "42\n";
    }
};

class Derived: public Base
{
public:
    void show_number() const override
    {
        std::cout << "101\n";
    }
};

void some_function(Base thing_showing_number)
{
    thing_showing_number.show_number();
}

int main()
{
    Derived derived{};
    some_function(derived);
}
```

Cheat if you want, and try [Godbolt](#). So, what happens, and more importantly, why? You get 42 output, even though you had a Derived class, whose function displays 101. Did you notice `some_function` takes the object by value? Any object that is a Base, including derived types, can be passed to the function. But, the derived part is chopped off, called *slicing*: only the base subobject is copied. You therefore end up using

the base's virtual functions, rather than the derived virtual functions. Slicing is almost always annoying. You need to change the function to take a reference:

```
void some_function(Base & thing_showing_number)
```

You then get 101 output, because the reference uses the right type. But, you already know if you want to use OOP you have to use references or, better, smart pointers. Furthermore, keeping your base class abstract avoids slicing. You can't create an abstract class, so you can't slice to an abstract base.

Conclusion

You learned how to write a base class, ensuring you provide a virtual destructor. You used pure virtual methods in your base class, making it abstract. You use the keyword `virtual` to indicate the function is designed to be overridden. You make it pure by adding `=0` to indicate there is no implementation. You used type traits again along with `static_assert` to test the class is in fact abstract.

You rewrote your trading game to use the `Exchange` class from [Chapter 12](#), using an `Asset`. You wrote two concrete classes, derived from the abstract base `Asset`, so could use either derived type in the game. The choice of type was dynamic: happening at runtime.

You learned to use `override` to ensure you get a function signature correct in a derived class. An overridden function replaces the base class function. One of your derived classes generated random prices and the other used a `std::vector` of prices. The `Exchange` itself didn't need to change when you added the second derived class. Polymorphism allows you to change behavior without altering existing code.

You also wrote your own exception, deriving from `std::exception`. You can derive from any type with a virtual destructor. In the deep dive, you learned about how virtual functions might be implemented and why a virtual destructor is important.

[Chapter 14](#) will show you another way to vary behavior according to type, without using OOP.

Using `std::variant` and `std::visit`

You have seen how to vary behavior with classes, starting with an abstract base class to provide an interface. You can then add as many derived classes as you need to provide varying implementations. I took a few chapters to teach you all the building blocks to do this in C++. There are several parts you need, but the approach is extensible: you can continue to add extra derived types as you need them, without changing existing code.

You can use another C++ feature, called the `std::variant`, to vary behavior too. The `std::variant` is a class template introduced in C++17. The `std::variant` holds one of many alternative types, which C programmers may recognize as an approach similar to a union. The types can be completely unrelated. This approach requires a fixed set of types up front, so it isn't as extensible as OOP. However, `std::variant` allows you to extend the behavior of the set of types unintrusively. If you want to add functionality to classes, you can add a new method in the base class, which will affect all the derived classes. If you use a `std::variant`, you can provide new functions without changing the types the `std::variant` contains. OOP makes it easy to add new types, but hard to add new operations. Variants make it easy to add new operations but hard to add new types, so OOP and variants have trade-offs.

This chapter will show you how to use a `std::variant` to add potential bonus payments or fines in your trading game: for example, an interest payment. You will also meet two other features, `std::optional` and `std::any`, that also hold varying types but have different design goals.

Creating and Using a `std::variant`

Let's add some extra possibilities to your trading game. You could have a news headline, which can be a `std::string`, and create other types representing a fine, a gift, or an interest payment. You will create a new function that sometimes returns one of these at random. The game will report if an event happened, either showing the `std::string` or displaying a message and adjusting the funds accordingly.

How do you return one of many unrelated types? If you have a base class, you can return a smart pointer to that class, and polymorphic behavior will take care of things. For unrelated types, that is not possible. You can, however, put any types in a `std::variant`.

Let's start with some new types for fines, gifts, or interest payments in a new *events.h* file. You only want an event to happen occasionally, so you can define a nonevent too. You then use the events and the nonevent in a `std::variant`:

```
#include <iostream>
#include <string>
#include <variant>

namespace stock_prices
{
    struct Nothing ❶
    {
    };

    struct FixedFine ❷
    {
        double fine{};
    };

    struct Gift ❸
    {
        double gift{};
    };

    struct InterestPayment ❹
    {
        double percent{};
    };

    using Event = std::variant<Nothing,
        FixedFine,
        Gift,
        InterestPayment,
        std::string>; ❺
}
```

- ❶ Defines a nonevent
- ❷ Defines a fixed fine, which will be subtracted from the funds
- ❸ Defines a gift, which will be added to the funds
- ❹ Defines an interest payment, which will also be added to the funds
- ❺ Declares an Event type, which is a `std::variant` capable of holding the new types, or a `std::string`, which holds a message

You will use the Event in your trading game shortly. Let's find out how to create and use the `std::variant` first. Make a new *main.cpp* file to experiment and include your new header.

You can use `std::holds_alternative` to see if a variant holds a value for a specific type:

```
#include <iostream>

#include "events.h"

int main()
{
    Event event{FixedFine{10.0}}; ❶
    if(std::holds_alternative<FixedFine>(event)) ❷
    {
        std::cout << "A fine\n";
    }
}
```

- ❶ Declares an Event of a FixedFine with value 10.0
- ❷ Checks if the Event contains a FixedFine

You will see A fine output. You use `std::get` to obtain the value. You can request a specific type or use an index corresponding to the position in the `std::variant`. Recall the definition:

```
using Event =
    std::variant<Nothing, ❶
                FixedFine, ❷
                Gift, ❸
                InterestPayment, ❹
                std::string>; ❺
```


- ❶ Places `Nothing` at index 0
- ❷ Places `FixedFine` at index 1
- ❸ Places `Gift` at index 2
- ❹ Places `InterestPayment` at index 3
- ❺ Places a message at index 4

You can get the `FixedFine`, which is at index 1, like this:

```
FixedFine fixed_fine_by_type = std::get<FixedFine>(event); ❶
std::cout << fixed_fine_by_type.fine << '\n';
FixedFine fixed_fine_by_index = std::get<1>(event); ❷
std::cout << fixed_fine_by_index.fine << '\n';
```

- ❶ Gets the `FixedFine` by type
- ❷ Gets the `FixedFine` by index

`std::get` is a template, so you provide the type or index as the template parameter. If the `std::variant` doesn't hold the type specified, or a different index is in use, you get a `std::bad_variant_access` exception thrown.

By default, the first type is populated:

```
Event another_event{}; ❶
if(std::holds_alternative<Nothing>(another_event)) ❷
{
    std::cout << "Nothing\n";
}
```

- ❶ Declares a `std::variant` without specifying a value
- ❷ Holds the first alternative by default

Since the `std::variant` can hold any one of the declared types, you can change the value:

```
another_event = InterestPayment{0.04}; ❶
if(std::holds_alternative<InterestPayment>(another_event)) ❷
{
    std::cout << "An interest payment\n";
    std::cout << "interest " <<
        std::get<InterestPayment>(another_event).percent ❸
        << '\n';
}
```

- ❶ Sets the event to a different type (and value)
- ❷ Checks which alternative the event now holds
- ❸ Gets the value, via the type

Now, checking which alternative is in play and getting the value are slightly cumbersome. If someone changes the order of the types, you have a maintenance headache! C++ provides a way to operate on the type a `std::variant` holds without having to check the alternative explicitly: via `std::visit`.

Using a `std::variant` in `std::visit`

The simplest way to use `std::visit` is with a class providing overloaded call operators for each type in a `std::variant`. You met operator `()`, the call operator, in [Chapter 5](#). There you used `std::greater`, whose call operator compares two values. `std::visit` will apply the operator to one value: that held by the variant. The Event has five possible types:

- Nothing
- FixedFine
- Gift
- InterestPayment
- `std::string`

The event visitor therefore needs five overloaded call operators. The compiler will tell you if your visitor doesn't have an overload for one of the variant's types. Nothing does nothing, the `std::string` is for a message, and the others alter the funds. Add a new struct, `EventVisitor`, to your *events.h* file:

```
struct EventVisitor
{
    double & funds; ❶
    void operator()(Nothing) {}; ❷
    void operator()(const FixedFine & event) ❸
    {
        std::cout << "Fixed fine " << event.fine << '\n'; ❸
        funds -= event.fine; ❸
    }
    void operator()(const Gift & event) ❹
    {
        std::cout << "Gift " << event.gift << '\n'; ❹
        funds += event.gift; ❹
    }
}
```

```

void operator()(const InterestPayment & event) ❸
{
    std::cout << "Interest payment " << event.percent << "%\n"; ❺
    funds *= (1.0 + event.percent); ❺
}
void operator()(const std::string & message) ❻
{
    std::cout << message << '\n'; ❻
}
};

```

- ❶ Refers to funds
- ❷ Does nothing
- ❸ Charges a fine
- ❹ Gives a gift
- ❺ Adds a percentage interest payment
- ❻ Displays a message

You also need to include `<iostream>` for `std::cout`. Notice the reference to `funds`, which are in your trading game. The game code will provide the actual value to refer to, and you need to ensure it stays in scope while your `EventVisitor` wants to use the reference.



I warned you about dangling references in “[Lambda Captures by Reference](#)” on page 127. Whenever you use a reference, you need to ensure that the object it refers to does not go out of scope. The *lifetime*, or scope, of the object referred to must be longer than the lifetime of the reference. Some C++ tools offer a flag called *address sanitizers*, which can check for dangling references. Recent versions of Microsoft’s Visual Studio have an **address sanitizer**, which you can enable in the C++ general setting in your project. **Clang** and **GCC** also provide address sanitizers. Such lifetime issues are hard to spot, so be careful when using member variables that are references.

You can now visit your Event by passing an `EventVisitor` to `std::visit`, along with the Event:

```

#include <iostream>

#include "events.h"

int main()
{
    Event event{ Gift{ 25.00 } }; ❶
    double funds{ 0.0 }; ❷
    std::visit(EventVisitor{funds}, event); ❸
    std::cout << "funds " << funds << '\n'; ❹
}

```

- ❶ Creates a Gift event
- ❷ Declares some funds
- ❸ Visits the Event
- ❹ Reports the funds

The funds start at 0; then the `EventVisitor` applies the gift, selecting the operator taking a Gift. The reference to funds in the `EventVisitor` increases the funds by the gift amount, so you see increased funds:

```
funds 25
```

You can also provide a message as an Event:

```
std::visit(EventVisitor{ funds }, Event{ "You won a free magazine" });
```

The operator taking a `std::string` doesn't affect the funds, just displays the message:

```
You won a free magazine
```

You have added behavior based on the type, but you don't need a base class to relate the `std::variant`'s types together. The type in play determines the behavior `std::visit` selects. You could write a different visitor to add more behavior. You are free to add new operations when you use a visitor to a `std::variant`. (This is something you cannot easily do in an OOP setting. In OOP you would need to add a new virtual function in a base class—which would be highly disruptive, because all users of the base class would have to update code.)

Let's write a function to return a random Event and then use the new function in your trading game.

Using the Event in the Trading Game

You can make a `std::array` of Events and randomly select one. Rather than letting each Event be equally likely, which a uniform distribution gives you, you can make some selections more likely than others using a `discrete_distribution`. Add the definition to a new `events.cpp` file (and don't forget to declare the function in `events.h`), as shown in [Example 14-1](#).

Example 14-1. Randomly picking an Event

```
#include <array>
#include <random>

#include "events.h"

stock_prices::Event stock_prices::generate_event()
{
    static std::mt19937 engine{ std::random_device{}() };❶
    std::array<Event, 5> events{ Nothing{},
        FixedFine{2.5},
        Gift{25.00},
        InterestPayment{0.04},
        "You won a free magazine"};❷
    std::discrete_distribution<> dist({ 60, 10, 10, 10, 10 });❸

    return events[dist(engine)];❹
}
```

- ❶ Declares a static random number engine, seeded by `std::random_device`
- ❷ Creates a `std::array` of five Events
- ❸ Assigns probabilities to the Events: 60% for Nothing and 10% for each other Event
- ❹ Returns the randomly selected Event

Did you notice the keyword `static` in the new function? The Mersenne Twister engine, which you met in [Chapter 7](#), is quite large, so creating one every time you call the function is expensive. If you mark a variable as `static`, it is created the first time the function is called and lives until your program finishes. An alternative would be to create the engine in `main` and pass it to the `generate_event` function. I showed you `static` because it does get used from time to time: for example, for caching large objects or ones that are expensive to make. However, a `static` variable isn't obvious outside the function: in effect, you have introduced hidden global state. This can

make code hard to reason through, and the state will persist between calls, which can make testing difficult. I think `static` is mostly best avoided, but it can be useful.

I have also introduced a different random distribution: the `discrete_distribution`. A uniform distribution makes each number equally likely. The *discrete* distribution lets you *weight* the values, that is, make some more likely than others. When you call `dist(engine)`, you are likely to get 0 60% of the time and 1, 2, 3, or 4 10% of the time. You use the random selection as an index into the `std::array` to pick an Event.

You can call `generate_event` in your trading game. Find your *trade.cpp* file and include the *events.h* header. Call the `EventVisitor` for a randomly generated Event:

```
#include <cassert>
#include <format>
#include <iostream>
#include <print>

#include "events.h" ❶
#include "trade.h"

double stock_prices::trading_game(Exchange & exchange)
{
    const double initial_funds{ 100.0 };
    double funds{ initial_funds };
    int number_of_shares{};

    bool playing{ true };
    while(playing)
    {
        auto status = std::format("Funds ${:.2f}, Shares {}",
                                  funds, number_of_shares);
        std::println("{} ", status);
        const auto price = exchange.next_price();
        auto price_message = std::format("Current price: ${:.2f}", price);
        std::println("{: >{}}", price_message, status.size());
        std::println("Press (s) to sell, (b) to buy, (q) to quit");
        std::print("or something else to continue>");
        char choice{};
        std::cin >> choice;

        if (choice == 's')
        {
            if (number_of_shares > 0)
            {
                exchange.fulfill_sell_order();
                --number_of_shares;
                funds += price;
            }
            else
            {
                std::println("No stock to sell");
            }
        }
    }
}
```

```

    }
}
else if (choice == 'b')
{
    if (price <= funds)
    {
        exchange.fulfill_buy_order();
        ++number_of_shares;
        funds -= price;
    }
    else
    {
        std::println("Insufficient funds");
    }
}
else if (choice == 'q')
{
    playing = false;
}
std::visit(EventVisitor{funds}, generate_event());❷
}
return funds - initial_funds;
}

```

❶ Includes the event header

❷ Generates a random event

You can use the `main` from [Example 13-5](#) to call the new version of your trading game. Don't forget to add `events.cpp` to your build. You might see a random event from time to time:

```

Funds $100.00, Shares 0
  Current price: $4.89
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>b
You won a free magazine ❶
Funds $95.11, Shares 1
  Current price: $4.90
Press (s) to sell, (b) to buy, (q) to quit
or something else to continue>n
Interest payment 0.04% ❷
Funds $98.92, Shares 1
  Current price: $4.93

```

❶ Shows a message

❷ An interest payment was made

An Event is chosen after each price update, but most of the time Nothing is generated, so no message is displayed. Let's look in more depth at how `std::variant` works.

The `std::variant` in Depth

Some programming languages provide a type that can hold one of many types. Unfortunately, you need to keep track of which type is in play, which can lead to mistakes. The `std::variant` is similar, but it's *type-safe*: if you try to access the wrong type, you get an exception. Type safety is an important feature in C++. It either detects problems at runtime or, better yet, gives a compilation error.

When you used smart pointers, you had to dereference the pointer to access the value. In contrast, you can use a `std::variant`'s value directly, giving you *value semantics*: programming that focuses on values rather than objects. Value semantics is a different way of coding than using pointers or references, which is called *reference semantics*. C++ supports various programming paradigms and styles. [Andrzej Krzemiński's blog](#) is a good starting point for further details on value semantics. Sean Parent and Dave Abrahams have also talked about value semantics for decades. Dave Abrahams' talk from [CppCon 2022](#) gives more details, and Klaus Iglberger's [C++ + Software Design](#) is another excellent resource.

Coding with a `std::variant` is simpler than using a class hierarchy, but you need a closed set of types for the `std::variant`. When you write a base class, you can add new derived types without needing to change the code by using the interface defined by the base class. If you want to add a new Event, you need to add the new type to the variant, so all code that uses it needs to recompile. So, smart pointers and variants have trade-offs.

A `std::variant` can be relatively large compared to a smart pointer or to using a single type directly. The `std::variant` uses enough memory for the biggest type it can hold, even if it holds a smaller type. If you used a Gift directly, you would use less space. Sometimes larger objects make your code slower. [Figure 14-1](#) illustrates a `std::variant`'s size.

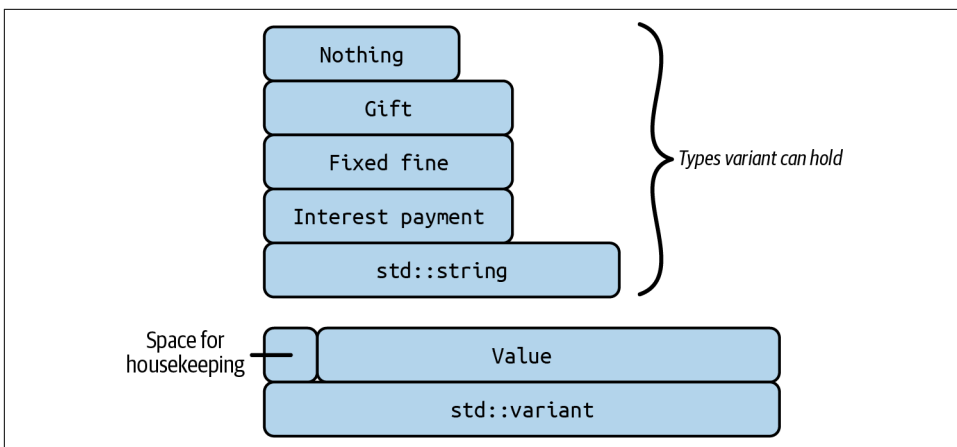


Figure 14-1. A `std::variant` is at least as large as the biggest type it can hold

Some people use a `std::variant` instead of a class hierarchy, but needing a fixed set of types up front means you lose some flexibility. Nicolai Josuttis gave a talk called “Rethink Polymorphism in C++” at [C++ on Sea](#) in 2025, exploring the pros and cons. You can watch it on [YouTube](#).

Spotting and Handling Potential Problems with `std::variant`

I mentioned that the `std::variant` will throw a `std::bad_variant_access` if you try to get the wrong value. Let’s look at an example and think about other potential problems and edge cases.

You can ask for a specific type:

```
Event event{ Gift{ 25.00 } };
auto gift = std::get<Gift>(event);
```

You can also ask for an index. Which index is `Gift`? Like me, you probably need a reminder:

```
std::variant<Nothing, FixedFine, Gift, InterestPayment, std::string>;
```

What happens if you get the wrong index? Maybe you use an index that’s too large:

```
auto gift = std::get<6>(event);
```

You get a `static_assert` telling you the index is too large, so the code won’t compile. You could use an index that’s within range, but still wrong:

```
Event event{ Gift{ 25.00 } }; ❶
auto gift = std::get<1>(event); ❷
```

❶ Sets a Gift, which is index 2

❷ Tries to get index 1

Index 1 refers to a FixedFine, but the `std::variant` holds a Gift, so a `std::bad_variant_access` is thrown. Needing to remember which index is which is annoying. Using the type is less error-prone, but you can have the same type more than once!

You can use `std::get_if` instead of `std::get`:

```
Event event{ FixedFine{ 100.0 } };  
auto gift_pointer_from_type = std::get_if<Gift>(&event); ❶  
auto gift_pointer_from_index = std::get_if<2>(&event); ❷
```

❶ Tries to get a Gift using the type

❷ Tries to get a Gift using an index

Notice that `std::get_if` takes a pointer to a `std::variant` and returns a pointer. If you run the code, both `gift_pointer_from_type` and `gift_pointer_from_index` will be a `nullptr`, because you set the Event to a FixedFine this time. You therefore need to check that you actually got a value and are back to using pointers.

The edge cases are important to know, but when you use `std::visit`, you don't need to access specific values manually. Even better, the `std::visit` warns you if you miss a type in the visitor.

Using `std::optional` and `std::any`

C++ has a couple of other features that allow you to have a variable that can hold various types. The `std::variant` holds one of many values. Sometimes you might want to indicate that a value isn't set. Maybe you have a value of a specific type, or maybe you have nothing. `std::optional`, from the `<optional>` header, provides this functionality.

You state the type you may have:

```
std::optional<char> choice{'b'}; ❶  
if (choice) ❷  
{  
    std::cout << "something\n"; ❸  
}  
else  
{  
    std::cout << "nothing\n"; ❹  
}
```

- ❶ Optionally has a char, set to 'b' here
- ❷ Checks if you have a value
- ❸ Shows you have something
- ❹ Shows you have nothing

In this case the choice is a character, so you see something output.

You can make the optional not have a value:

```
std::optional<char> choice{};
```

In this case, you see nothing.

You can get the value from a `std::optional`:

```
std::cout << "character" <<
  choice.value() ❶
  << '\n';
```

- ❶ Calls `value` to access the `std::optional`

Calling `value` when there is none throws a `std::bad_optional_access`. You met `std::expected` in [Chapter 3](#). That provided a way to return an expected value or an error. You can use the `std::optional` in a similar way, providing a value or not. The calling code can then decide how to proceed if there is no value.

The next type you can use to hold any type is called `any`, defined in the `<any>` header. You use it like this:

```
std::any value = 42;
```

`std::any` can also be empty. For historical reasons, the functions for `std::any` don't follow the `get` functions for the `std::variant`. You use an `any_cast` for the type to get the value, like this:

```
std::any_cast<int>(value);
```

There may not be a value, or the type held might not be an `int`, in which case you get a `std::bad_any_cast` thrown. The `reset` function sets a `std::any` to nothing. You can check you have a value first, but you still need to remember what type you're after:

```
value.reset(); ❶
if (value.has_value()) ❷
{
    std::cout << "value " <<
      std::any_cast<int>(value) ❸
      << '\n';
```

```

    }
    else
    {
        std::cout << "nothing\n"; ❹
    }

```

- ❶ Sets the value to empty
- ❷ Checks if the `std::any` has a value
- ❸ Tries to get an `int`
- ❹ Reports that the value is empty

Since the value was reset, this code reports nothing.

Conclusion

You met `std::variant` in this chapter and used `std::visit` to apply a function to the type held. The `std::variant` requires a fixed set of types up front, so it doesn't provide the extensibility of OOP, but you don't need a base class or smart pointers.

You used `std::holds_alternative` to check if a `std::variant` had a specific type in play. You also used `std::get` and `std::get_if` to obtain the values. Both approaches let you use a type or an index. `std::get` throws a `std::bad_variant_access` if you try to get the wrong type or index. `std::get_if` takes a pointer to a `std::variant` and returns a pointer. If you get `nullptr` back, the `std::variant` either had no value or holds a different type, or else a different index is in play.

You can use `std::visit` to add new behavior to the types in a `std::variant`. You write a visitor with an overload for each type and then don't need to call the getters. The compiler will also tell you if your visitor doesn't have an overload for one of the variant's types.

You also met `std::optional` and `std::any`, which are other ways to hold varying types, with different (more restrictive, but simpler) use cases.

And you used the keyword `static` to keep a variable in a function between calls. I showed you how to use the `std::discrete_distribution` when you want to pick values at random, making some more likely than others.

The next chapter is the last. You've learned a lot so far, but there's more to learn. There isn't enough space to cover everything, but you will be well placed to read and write C++ when you finish this book.

Templates and `std::unordered_map`

In this final chapter, you will use a lookup table to track how many of each Event type happens in your game. You will tally the frequency of each Event during the game and report back afterward. The tally won't add anything to the game itself, but it will show you another useful C++ container. You will use a `std::unordered_map` to keep the frequencies per Event, and that means you need to write your own template to facilitate the lookup. Adding lookup tables to your repertoire and knowing how to write templates will leave you with a firm grounding in a range of C++. You can use the tally to check that you get the different Events with the probabilities you requested.

You will also learn about defaulting the equality operator for your types and about the `std::pair`, and you'll get the chance to write another visitor. You've covered a lot of C++ now, so let's finish up with a few last details. I haven't covered everything—C++ is a big language—but you know enough to write a whole program, and you know where to look things up.

Making a Lookup Table

Lookup tables are sometimes called *dictionaries* in other languages. They are a type of *associative container*: a container that provides fast lookup. A lookup table contains key-value pairs. Your key will be an Event, and the value will be the tally.

C++ provides two types that map unique keys to unique values. The older container is the `std::map`, defined in the `<map>` header. This container uses a comparison to order the elements, which makes searching quicker than, say, iterating through an unordered `std::vector`. The newest is the `std::unordered_map`, defined in the `<unordered_map>` header. The unordered version uses a different approach to speed up searching. Let's look at the details.

You need to provide a key and a value for a `std::unordered_map`, in that order:

```
std::unordered_map<std::string, int> lookup;
```

This lookup maps a `std::string` to an `int`. You can use operator `[]` to get and set values:

```
lookup["Hello"] = 1; ❶  
int count = lookup["Hello"]; ❷
```

❶ Sets the key "Hello" to the value 1

❷ Gets the value for key "Hello"

If the value isn't there, you get a default key. The operator `[]` actually inserts a value in this case. Not having to check whether the key exists first can be useful. For example, you get zero if you try to get a nonexistent item for this lookup:

```
int count = lookup["Goodbye"]; ❶
```

❶ Gets a count of 0

However, be aware that trying to get a value might have the hidden cost of setting a value for you.

If you use built-in types, you only need to provide the key and value types. You're going to build a lookup for Events, though, so you need to provide a hash as well. A *hash* is a function that returns a numeric value for an object. The `std::vector` stores elements contiguously. If you want to find one, you might have to iterate through the whole `std::vector`. The `std::unordered_map` stores elements in buckets instead. The key's hash dictates into which bucket the key-value pair goes. This means that the operator `[]` can go straight to the right bucket to look for your key-value pair.



A good hash gives a unique bucket index for distinct elements. If two distinct elements end up with the same hash, you have a *clash*: the container can no longer jump straight to one element. It needs to check each element in a bucket to find the one you want. You don't need to worry about this too much, but it's worth being aware of. CppReference gives details on [checking how many buckets you have](#) so you can detect clashes. If you have fewer buckets than elements, it means some elements are sharing a bucket.

C++ has a `std::hash` for `int`, `std::string`, and most other standard library types. Again, CppReference provides details. There is a `std::hash` for `std::variant`, which uses the hash for each type it contains. Event is a `std::variant`, but you don't have hash functions for all the Event types yet. The Nothing, Fine, Gift, and

InterestPayment need hash functions. Now, `std::hash` is a template, so the easiest way to add a hash for your Event is to make `std::hash` for these Event types.

Let's learn about writing templates first, and then I'll show you how to make `std::hash` work for an Event, so you can make a tally of Events.

Write Your Own Template

You can make a *function template* or a *class template*. Your template defines how to make functions or classes for specific types or values. You've used `std::vector` several times now. The class template describes how to make a vector for any type, so `std::vector<int>` uses the template for an `int`. You also used a `std::array`, which takes a type and a value, like this: `std::array<int, 5>`.

Let's look at function templates first. You'll write a class template shortly.

A function template looks like an ordinary function but has an extra part first, the *template parameter list*:

```
template<typename T> ❶
void function(T value) ❷
{
    std::cout << value << '\n';
}
```

- ❶ Declares a template for a type `T`, with a template parameter list
- ❷ Defines a function taking any type `T`

You'll sometimes see the word `class` instead of `typename`: either can be used in the template parameter list. People often use `T` for the type, but you can use any name you like (provided it's not a keyword).

No code is generated until you use the function for a specific type. To call the function, you can specify the type in `<>`, as you have done for `std::vector` and many other standard library types, or rely on *template argument deduction*, where the compiler deduces the type based on the given argument:

```
function<int>(10); ❶
function(10); ❷
```

- ❶ Explicitly states you are calling the function for an `int`
- ❷ Uses template argument deduction

Recall “Using a Lambda to Vary Behavior via `std::function`” on page 111, where you passed a lambda to a function taking a `std::function`? That allowed you to vary a prompt when you got some prices:

```
std::vector<double> get_prices(std::istream & input_stream,
                             std::function<void ()> prompt);
```

I warned you that lambdas get copied into the `std::function`, so they aren’t as efficient as possible. You can change the function to be a function template instead and avoid the copy. You need to put the code in the header, because it needs to be fully visible to generate code for the template parameter, as shown in [Example 15-1](#).

Example 15-1. A template for the prompt

```
#pragma once

#include <expected>
#include <istream>
#include <string>
#include <vector>

namespace stock_prices
{
    std::expected<double, std::string> get_number(std::istream & input_stream);

    template<typename T>
    std::vector<double> get_prices(std::istream & input_stream,
                                  T prompt) ❶
    {
        prompt(); ❷
        std::vector<double> numbers{};
        auto number = stock_prices::get_number(input_stream);
        while(number.has_value())
        {
            numbers.push_back(number.value());
            prompt(); ❸
            number = stock_prices::get_number(input_stream);
        }
        return numbers;
    }

    void test_input();
}
```

❶ Takes a prompt as a template parameter

❷ Calls the prompt

❸ Calls the prompt again

Now the lambda can be used directly, and you don't need to change the calling code.

Class templates are similar. Again, you provide a template parameter list, and you can use the type throughout your class:

```
template<typename T> ❶
class Structure ❶
{
    T value{}; ❷
public:
    explicit Structure(T value) ❸
        : value(value)
    {
    }
    void member_function() const ❹
    {
        std::cout << value << '\n';
    }
};
```

- ❶ Declares a class template for type T
- ❷ Declares a member variable of type T
- ❸ Provides an explicit constructor taking the templated type by value
- ❹ Uses the value in a member function

You can create a `Structure`, and *class template argument deduction* (CTAD) will deduce the type for T:

```
Structure structure{101};
structure.member_function();
```

You could explicitly state the type (`Structure<int>`), too, but you don't need to.

The template is a way to generate code. You therefore need the *whole* template, both declarations and definitions, to be visible when you use it, so put your templates in header files. You can't split the implementation into a source file. Nothing is added to your program until you use the template. This means compiler errors might not be reported until you try to use the template.

Speaking of nothing: try the `Structure` for the `Nothing` Event:

```
Structure nothing_structure{Nothing{}};
```

This compiles OK, but now try to call the member function:

```
nothing_structure.member_function();
```

You will see a lot of errors, maybe starting with something like:

```
error: no match for 'operator<<'
(operand types are 'std::ostream'
 {aka 'std::basic_ostream<char>'} and 'const Nothing')
std::cout << value << '\n';
```



Don't panic if you get a lot of errors from your compiler. Each error specifies a line in a file and may state problems using library functions. Find an error that relates to your code and work from there. Is a function missing? Does it have the wrong signature? Have you got a typo?

Member functions of class templates use *lazy instantiation*, where the member function is created only when it is used. If you don't use the member function, no code is generated for the function. That's useful, because it doesn't waste space, but it can lead to surprises. The `member_function` uses operator `<<` for the type, but you haven't written one for `Nothing`, so you get an error.

You can overcome this problem by *specializing* your template: giving a different implementation for a specific type after the original template definition. You leave the template parameter list empty (`template<>`) and put your specialization type after the class name:

```
template<> ❶
class Structure<Nothing> ❷
{
public:
    explicit Structure(Nothing value) ❸
    {

    }
    void member_function() const ❹
    {
        std::cout << "Nothing\n";
    }
};
```

- ❶ Empty template parameter list
- ❷ Explicit specialization of `Structure` for `Nothing`
- ❸ Defines a constructor, but you don't need to save the value this time
- ❹ The member function outputs `Nothing` and a new line

Now you can make a Structure from Nothing and call the `member_function`:

```
Structure nothing_structure{Nothing{}};  
nothing_structure.member_function();
```

The specialization of `Structure` is selected, and you see `Nothing` output. Notice that the data member value is missing. That's OK. A class template specialization is a completely different type, so you don't need this data member here.

Now, you want a tally of Events, so you want a `std::unordered_map` of Event keys to `int` values. Try to declare a lookup for a tally:

```
std::unordered_map<Event, int> lookup;
```

The template will fail to compile, and you'll get a lot of errors mentioning deleted functions and hash tables. Don't be put off. Somewhere near the end, you will see something like:

```
static assertion failed  
static_assert(noexcept(declval<const __hash_code_base_access&>  
...)
```

The `std::unordered_map` fails to compile for `Event`, much like `Structure` failed to compile for `Nothing`. The key in a `std::unordered_map` needs to have a hash function and an equality comparison, so let's see how to provide these for your `Event` using what you just learned about templates and specializations.

Specializing `std::hash`

I told you a `std::unordered_map` uses a hash function, returning a numeric value for an object. This dictates into which bucket an object goes. If you have more than one object in a bucket, the `std::unordered_map` needs a way to decide if any two objects are equal. The `std::unordered_map` defaults the hash and equality like this:

```
template<  
    typename Key, ❶  
    typename T, ❷  
    typename Hash = std::hash<Key>, ❸  
    typename KeyEqual = std::equal_to<Key>, ❹  
    typename Allocator = std::allocator<std::pair<const Key, T>> ❺  
> class unordered_map;
```

- ❶ Defines the type of the keys in the map.
- ❷ Defines the type of the values associated with the key.
- ❸ Defaults to `std::hash` for the `Key`, to decide which bucket to use.

- ④ Defaults to `std::equal_to` to see if keys are equal, in case more than one object ends up in a bucket.
- ⑤ Provides an allocator, which defines where elements are put. They go on the heap by default.

The last three template parameters have defaults. I've not mentioned allocators yet. All the containers allow you to change where elements are allocated. Allocators are an advanced topic, and providing the key, value type, and hash is enough often enough. Patrice Roy's *C++ Memory Management* is useful if you want to know more about allocators. The `std::equal_to` gets used if several elements end up in one bucket, and defaults to calling operator `==`. You got the `static_assertion` failure in the last section because there is no `std::hash` for `Event` yet.

Your `Event` needs both a hash and a way to check for equality via operator `==`. Let's sort out the hash first.

You can provide the hash function in various ways, but it's simplest to make `std::hash` work for your `Event`. The `std::hash` is a class template:

```
template< typename Key > ①
struct hash; ②
```

- ① Template parameter list
- ② Declaration of the struct `hash`

The `std::hash` is a template for making a struct for the `Key` type, `std::hash<int>` is a struct calculating the hash for an `int`, and `std::hash<std::string>` is a struct calculating the hash for a `std::string`. The hash itself is provided via a member call operator. The operator returns a `size_t`, takes a `Key` (which will be the specific type), and is `const` and won't throw:

```
size_t operator()(const Key & key) const noexcept;
```

Thus, you need to provide specializations of `std::hash` for each possible `Event` type.

Find your *events.h* file and add the definitions after (and outside) your `stock_prices` namespace, in namespace `std`:

```
namespace std { ①
    using namespace stock_prices; ②
    template<> struct hash<Nothing> ③
    {
        size_t operator()(const Nothing &) const noexcept ④
        {
            return 0; ④
        }
    }
}
```

```

};
template<> struct hash<FixedFine> ❸
{
    size_t operator()(const FixedFine & f) const noexcept ❹
    {
        return std::hash<double>{}(f.fine); ❺
    }
};
template<> struct hash<Gift> ❷
{
    size_t operator()(const Gift & g) const noexcept ❸
    {
        return std::hash<double>{}(g.gift); ❹
    }
};
template<> struct hash<InterestPayment> ❹
{
    size_t operator()(const InterestPayment & i) const noexcept ❺
    {
        return std::hash<double>{}(i.percent); ❻
    }
};
}

```

- ❶ Opens the namespace `std`
- ❷ Makes `stock_prices` visible within the `std` namespace
- ❸ Defines a specialization of `std::hash` for `Nothing`
- ❹ Uses `0` as a hash for `Nothing`
- ❺ Defines a specialization of `std::hash` for a `FixedFine`
- ❻ Uses the `fine`'s value for a hash
- ❼ Defines a specialization of `std::hash` for a `Gift`
- ❽ Uses the `gift`'s value for a hash
- ❾ Defines a specialization of `std::hash` for an `InterestPayment`
- ❿ Uses the `interest`'s percentage for a hash



In general, you should never add anything to the namespace `std`. The `std::hash` is one of a very few places where you can add things. In most other cases, you should add code to your own namespace to avoid clashing with other people's code. People expect code in the namespace `std` to be part of standard C++, so even if you don't collide with existing code, you will confuse people.

You can create a `std::hash` for your type and find the hash value for an instance:

```
#include <iostream>

#include "events.h"

int main()
{
    auto hasher = std::hash<stock_prices::Nothing>{}; ❶
    std::cout << hasher(stock_prices::Nothing{}) << '\n'; ❷
}
```

- ❶ Creates a hash for `Nothing`
- ❷ Outputs the hash value of a `Nothing`

You will get 0, since that is returned for any `Nothing` instance. In general, you want different objects to have a different hash, to avoid a clash. `Nothing` signifies no event, so in this special case it's OK to use the same value for each object. You do want a `Gift`, `FixedFine`, or `InterestPayment` with a different value to be treated differently and hence rely on `std::hash<double>` to do the right thing.

The `std::unordered_map` will use your hash when it needs a hash value to find a bucket. To make a tally of the `Events`, you need one more thing: a way to decide if two events are equal.

Adding an Equality Operator for an Event

You can declare a tally, but if you try to use it, you get compiler errors:

```
std::unordered_map<Event, int> tally; ❶
tally[Nothing{}] = 1; ❷
```

- ❶ Declares a tally, which is fine now that you have a `std::hash` for each event type
- ❷ Fails to compile at the moment

If you try to compile the code, you get several errors complaining about `key_equals`. Again, don't panic. You saw that a `std::unordered_map` uses `std::hash` and `std::equal_to` for the key by default. At the moment, you don't have a way to check if `Nothing`, `FixedFine`, `Gift`, and `InterestPayment` are equal. After the hash, a `std::unordered_map` defaults the way keys are compared for equality to `std::equal_to`:

```
typename KeyEqual = std::equal_to<Key>
```

The `std::equal_to` calls operator `==`, unless you specialize it for your type. So let's do that; you want to define operator `==` for each Event type. You can get C++ to generate the required code for you by adding a defaulted operator `==` to each struct:

```
struct Nothing
{
    bool operator==(const Nothing &) const = default; ❶
};

struct FixedFine
{
    double fine{};
    bool operator==(const FixedFine &) const = default; ❷
};

struct Gift
{
    double gift{};
    bool operator==(const Gift &) const = default; ❸
};

struct InterestPayment ❹
{
    double percent{};
    bool operator==(const InterestPayment &) const = default; ❺
};
```

- ❶ Requests a default operator `==` for `Nothing`
- ❷ Requests a default operator `==` for a `FixedFine`
- ❸ Requests a default operator `==` for a `Gift`
- ❹ Requests a default operator `==` for an `InterestPayment`

This feature was introduced in C++20. The default equality-comparison operator uses each data member in order to compare. Any member that also has members or is a container is recursively expanded in the comparison. That doesn't apply to your

classes: three have double data members, and `Nothing` has no members. For example, the `Fine` might be implemented like this:

```
bool operator==(const FixedFine& other) const
{
    return fine == other.fine;
}
```

If `Fine` contained other data members, they would be added to the resulting comparison. The `Gift` and `InterestPayment` are similar. `Nothing` has no data members, so each instance is equivalent. The implementation might look like this:

```
bool operator==(const Nothing& other) const
{
    return true;
}
```



You might also see default comparison operators defined as friend functions:

```
friend bool operator==(const Gift &, const Gift &) = default;
```

This means the function is defined outside the class, but the `friend` keyword means the function can see any data members. Scott Meyers's book *Effective C++* (3rd edition, Addison-Wesley, 2005, see Item 24) gives reasons why you might prefer to use a nonmember function here. For example, the nonmember version gives better encapsulation.

Now you can use the tally, because you have provided the hash and operator `==` required to make the get and set operator `[]` work. Let's add one more piece, and then you can use the tally in your trading game.

Adding a Way to Display the Events

Let's write a function to display the tally. You've used the range-based for loop to walk over containers before. A `std::unordered_map` is similar, but now you have a key and value at each iteration rather than a single element. There are a few ways to retrieve both, but the simplest is as follows:

```
for(auto [key, value] : tally) ❶
{
    // Use the key and value
}
```

❶ Gets the key and value

C++ is doing some magic for you here, which I will explain in “[Associative Containers and Templates in Depth](#)” on page 309. The important thing to note is that you have two variables that you can display, once you decide how to display the key.

The key is one of five events. You wrote an `EventVisitor` in “[Using a `std::variant` in `std::visit`](#)” on page 281 to update funds and display output. Add another visitor to your *events.h* file to display the event:

```
struct DisplayEventName
{
    void operator()(const Nothing &) ❶
    {
        std::cout << "Nothing"; ❶
    };
    void operator()(const FixedFine & event) ❷
    {
        std::cout << "Fixed fine " << event.fine; ❷
    }
    void operator()(const Gift & event) ❸
    {
        std::cout << "Gift " << event.gift; ❸
    }
    void operator()(const InterestPayment & event) ❹
    {
        std::cout << "Interest payment " << event.percent << "%"; ❹
    }
    void operator()(const std::string & message) ❺
    {
        std::cout << message; ❺
    }
};
```

- ❶ Displays Nothing
- ❷ Displays Fixed fine and the value
- ❸ Displays Gift and the value
- ❹ Displays Interest payment and the value
- ❺ Displays the message

There are other ways to display the events, like writing a function for each type that returns a string. However, using this way gave you some practice writing visitors. You can now add new functionality without needing to change the original classes. Neat!

Now you can call your display function. You will use this in your trading game, so put the code in *trade.cpp*:

```

#include <cassert>
#include <format>
#include <iostream>
#include <print>
#include <unordered_map> ❶

#include "events.h"
#include "trade.h"

namespace stock_prices ❷
{
    void display(const std::unordered_map<stock_prices::Event, int> & tally) ❸
    {
        for(auto [key, value] : tally) ❹
        {
            std::visit(stock_prices::DisplayEventName{}, key); ❺
            std::cout << " : " << value << '\n'; ❻
        }
    }
}

```

- ❶ Includes the `std::unordered_map` header
- ❷ Opens the `stock_prices` namespace
- ❸ Defines a `display` function taking your tally
- ❹ Iterates over the key and value pairs
- ❺ Visits the key to display it
- ❻ Displays the value directly

Declare the function in the *trade.h* file, too. Now you can use the tally in your game.

Keeping a Tally of Events in Your Trading Game

Let's add a tally to the trading game. You don't need to do much, because you've already written all the parts you need. You'll declare a tally, update it during the game, and then display it:

```

double stock_prices::trading_game(Exchange & exchange)
{
    const double initial_funds{ 100.0 };
    double funds{ initial_funds };
    int number_of_shares{};
    std::unordered_map<Event, int> tally; ❶

    bool playing{ true };
}

```

```

while(playing)
{
    auto status = std::format("Funds ${:.2f}, Shares {}",
        funds, number_of_shares);
    std::println("{} ", status);
    const auto price = exchange.next_price();
    auto price_message = std::format("Current price: ${:.2f}", price);
    std::println("{}: >{}", price_message, status.size());
    std::println("Press (s) to sell, (b) to buy, (q) to quit");
    std::print("or something else to continue>");
    char choice{};
    std::cin >> choice;

    if (choice == 's')
    {
        if (number_of_shares > 0)
        {
            exchange.fulfill_sell_order();
            --number_of_shares;
            funds += price;
        }
        else
        {
            std::println("No stock to sell");
        }
    }
    else if (choice == 'b')
    {
        if (price <= funds)
        {
            exchange.fulfill_buy_order();
            ++number_of_shares;
            funds -= price;
        }
        else
        {
            std::println("Insufficient funds");
        }
    }
    else if (choice == 'q')
    {
        playing = false;
    }
    const auto event = generate_event(); ❶
    std::visit(EventVisitor{funds}, event); ❷
    ++tally[event]; ❸
}
display(tally); ❹
return funds - initial_funds;
}

```

❶ Declares a tally

- ② Gets an Event
- ③ Displays the event as before
- ④ Updates the tally for this event, using ++ to preincrement the value
- ⑤ Displays all the events when the game is over

You can use the `main` from [Example 13-5](#) again to call the new version of your trading game. Toward the end of the game, you now see a tally of events:

```
Interest payment 0.04% : 4
Fixed fine 2.5 : 5
You won a free magazine : 5
Nothing : 22
Gift 25 : 4
```



The Gift doesn't say what currency it's in. It could even be magic beans rather than money! [Chapter 9](#) showed you how to format the number of decimal places displayed. You can add a dollar sign, but other currency symbols are non-ASCII, which makes displaying them harder and is beyond the scope of this book.

In [Example 14-1](#), you gave a nonevent (Nothing) a 60% chance of happening and everything else a 10% chance. The events are picked at random, so you might not get precisely those percentages, but the more you play the game, the closer they get. In this case, you have these percentages:

```
Interest payment 0.04%: 10%
Fixed fine 2.5: 12.5%
You won a free magazine: 12.5%
Nothing: 55%
Gift 25: 10%
```

They are relatively close to the discrete distribution you used. Creating a tally allowed you to check the percentages.

Did you notice that the order of the events doesn't match the order of the `std::variant`? The `std::variant` order is:

```
Nothing
FixedFine
Gift
InterestPayment
std::string
```

The organization of the `std::unordered_map` is based on the buckets, which are based on your hash, so you might not be able to guess the order. A

`std::unordered_map` is designed to make lookup quicker, which means the order is less important. Let's look in more detail and find out a bit more about templates.

Associative Containers and Templates in Depth

I showed you how to get key-value pairs from a `std::unordered_map`:

```
for(auto [key, value] : tally)
{
}
```

One tool you haven't met yet can help if you want to understand what code might be expanded to. **CppInsights** will generate details about code for you. You type in code on the left panel, labeled as "Source." For example, **Figure 15-1** shows the `main` function (the definitions are entered above and are not shown in this screenshot).

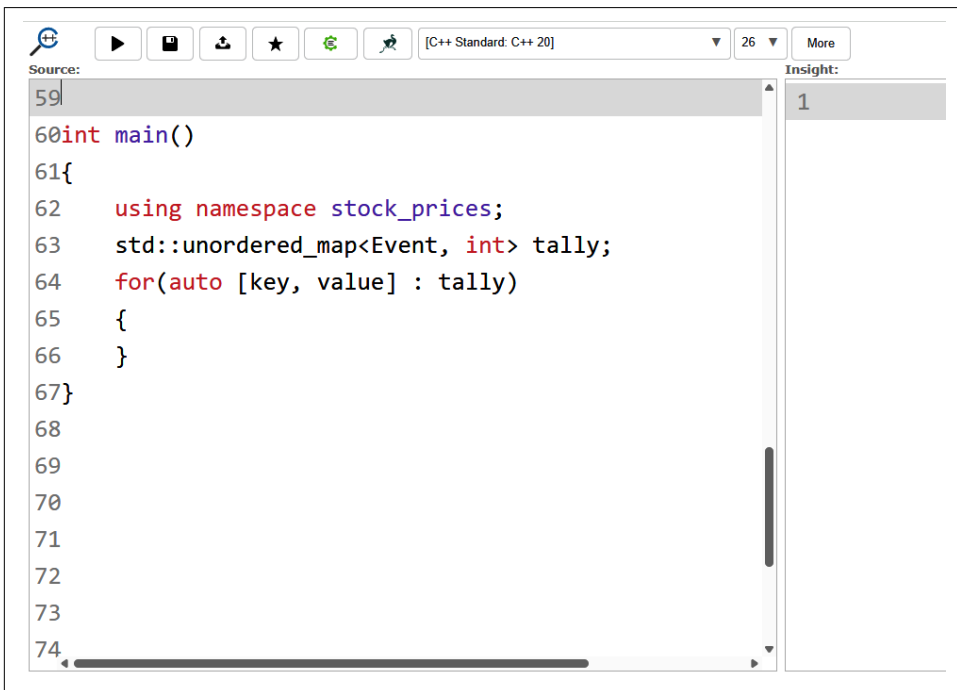


Figure 15-1. Some code entered into CppInsights

The full code is available at **CppInsights**. You then press the play button, on the top left, and the generated code is shown on the right. This is useful if your code is using syntactic sugar, discussed in **Chapter 2**, which may contain little more than punctuation. That can be hard to look up, so CppInsights at least tells you what you need to learn about.

The expansion of the loop over the tally has a lot of detail that won't fit across a page in this book! Nonetheless, the salient parts shows the for loop, something like this (some details omitted for brevity):

```
auto iterator = tally.begin();
auto end = tally.end();
for(; iterator != end; ++iterator) ❶
{
    std::pair<const std::variant<Event, int> __operator64 = *iterator; ❷

    const std::variant<Event> && key = std::get<0>(__operator64); ❸
    int && value = std::get<1>(__operator64); ❹
}
```

- ❶ Shows the range-based for loop as a C-style for loop
- ❷ Gets a `std::pair` from the iterator
- ❸ Gets the first item: the key
- ❹ Gets the second item: the value

You can see a `std::pair` of your `Event` (`std::variant`) and an `int`: the key and value of the tally. (The actual “Insight” shows more details, for example, spelling out `Event` in full as the `std::variant`.) You at least know the `auto [key, value]` involves a `std::pair`.

A `std::pair` holds two values of specific types. You guessed it: the `std::pair` is a template taking two types. The two values are stored in member variables called `first` and `second`. You can access the values using `first` and `second` like this:

```
auto first = tally.begin();
std::pair<const Event, int> first_pair = *first;
auto key = first_pair.first;
auto value = first_pair.second;
```

You can also use `std::get<0>` and `std::get<1>` to access the first and second values, as the Insight shows.

C++17 introduced *structured bindings*: a way to bind names to existing objects. Instead of getting the `std::pair` and the values from that, you directly bind to `first` and `second` with names of your choice.

As a reminder, here's the original code:

```
for(auto [key, value] : tally)
```

This binds `key` to `first` and `value` to `second`.

Using structured bindings is much simpler. You can even bind to a `std::array` or a simple structure. [CppReference](#) has further details. This is one of many ways in which C++ has become simpler recently.

More on Templates

I showed you how to write your own template in this chapter, but I'll own up: I already got you to write one earlier. In [“Custom Deleters” on page 244](#), you wrote a function to handle a pointer:

```
struct no_op_deleter
{
    void operator()(auto value) const ❶
    {
        std::cout << "Nothing to do for " << *value << "\n";
    }
};
```

- ❶ Declares an operator taking an auto

Using auto is shorthand for writing a template. The code is equivalent to:

```
struct no_op_deleter
{
    template<typename T> ❶
    void operator()(T value) const ❶
    {
        std::cout << "Nothing to do for " << *value << "\n";
    }
};
```

- ❶ Declares an operator template taking a type T

The version using auto means the same but is less to type. Once in a while, you might need to use the type T inside a class or function, so then you would need the full version with the template parameter list.

This `no_op_deleter` will fail to compile if the `value` isn't a pointer, or at least if it doesn't support dereferencing. You can add a *concept* before the `auto`: this checks for requirements on the type and can lead to better compiler error messages if the requirements aren't met.

You can use a type trait to check if something is a pointer and make a suitable concept for your `no_op_deleter`:

```
#include <type_traits>

template <class T>
concept pointer = ❶
```



```

std::is_pointer_v<T>; ❷

struct no_op_deleter
{
    void operator()(pointer auto value) const ❸
    {
        std::cout << "Nothing to do for " << *value << "\n";
    }
};

```

- ❶ Defines a concept
- ❷ Requires the type is a pointer
- ❸ States the template type must be a pointer

You can use the `no_op_deleter` as before, but what happens if you use it for something that isn't a pointer?

```

int value = 42;
std::unique_ptr<
    int,
    no_op_deleter
>
smart_pointer_to_value{ &value }; ❶
no_op_deleter{}(42); ❷

```

- ❶ Compiles fine, because you are using a pointer
- ❷ Fails to compile, because 42 is not a pointer

Notice that the operator in the `no_op_deleter` is templated.

You have now seen function and class templates, as well as templated member functions. Concepts are an advanced topic, but they can make compiler errors easier to understand. The `<concepts>` header has several predefined concepts you can use. Earlier, in [Example 15-1](#), you wrote a function template to use a prompt. You can constrain the template parameter with a concept to ensure the prompt is *invocable*: something that can be called, like a lambda or a function. You use `std::invocable<>` with an empty parameter list to mean a callable that takes no parameters and returns nothing:

```

#pragma once

#include <concepts> ❶
#include <expected>
#include <istream>
#include <string>
#include <vector>

```

```

namespace stock_prices
{
    std::expected<double, std::string> get_number(std::istream & input_stream);

    std::vector<double> get_prices(std::istream & input_stream,
        std::invocable<> auto prompt) ❷
    {
        prompt();
        std::vector<double> numbers{};
        auto number = stock_prices::get_number(input_stream);
        while(number.has_value())
        {
            numbers.push_back(number.value());
            prompt();
            number = stock_prices::get_number(input_stream);
        }
        return numbers;
    }

    void test_input();
}

```

- ❶ Includes the <concepts> header
- ❷ Uses `std::invocable<>` in the signature

If you try to call your `get_prices` function with an unsuitable parameter, the concept will add details to the error message.

Consider trying a lambda that takes an `int`:

```
stock_prices::get_prices(std::cin, [](int x) { return x+1; });
```

Compiler errors will vary, but g++ says:

```

note: template argument deduction/substitution failed:
note: constraints not satisfied

```

There are several other errors, but this tells you exactly what the problem is. Concepts are useful.

The templates you have considered in this chapter have all used a type. You can also have *nontype template parameters* (NTTP), such as a number. Remember meeting `std::array` in [Chapter 4](#)? That takes a type and a number:

```
std::array<double, 5> numbers{};
```

Nontype template parameters are another big topic. Being able to use nontypes, like numbers, means you can do sums with fractions using `std::ratio` at compile time, and more.

Conclusion

You have finished this book. Well done! You’ve rounded off by looking at a way to store key-value pairs in a lookup table, giving you a way to keep a tally of events. You used the `std::unordered_map` and provided a hash function and the operator `==` to decide where to store key-value pairs.

You learned about writing your own templates in detail, including how to specialize them for specific types. Since your key was a `std::variant`, you needed to write a `std::hash` specialization for your user-defined types before you could use the `std::unordered_map`. The `std::hash` is one of few places where you can add code to the namespace `std`. The organization of the `std::unordered_map` is based on the buckets, which are based on your hash, so you might not be able to guess their order. You requested a default operator `==` by adding the defaulted operator `==` to each type. For example:

```
bool operator==(const Nothing &) const = default;
```

This operator checks if the member variables (if there are any) are equal.

Each template is a way to generate code; it will be instantiated when you use it. You therefore need to put your templates in a header. You saw that you can use a type by writing `typename T` or `class T` in the template parameter list. Alternatively, you can use `auto`. You can also have nontype templates, such as a number, which `std::array` uses. You met concepts very briefly, which provide a way to constrain the type and can give clearer error messages.

C++ is a big language that is still evolving. There are also various online forums where you can get help. I personally appreciate the help from [ACCU](#). You can pay a small membership fee to join, but you can sign up to the general mailing list and ask questions there for free. ACCU are a friendly bunch, and I have learned so much from them. I also told you about [CppInsights](#). It’s a useful way to try to understand code, since it fills in some of the details. Don’t forget about [Godbolt](#) and [CppReference](#), too.

I think you will find that knowing some C++ helps you think about how different languages work, because C++ takes you closer to what is happening on your hardware. Knowing how to generate random numbers or objects underpins many games, so find another small game or project to write (perhaps rock, paper, scissors).

Keep on coding, but don’t panic if it doesn’t compile. Let the errors guide you. Write tests for your code; that will help you get it right. Keep asking questions and keep learning. Thank you for taking time to read this book. Above all, have fun and stay curious.

Symbols

&& operator, 20
-> operator, 238
:: operator, 19
<< operator, 13
== operator, 303
> (greater than), 52
>= (greater than or equal to), 52
>> operator, 16, 30
~ (tilde) operator, 170

A

a.out, 6
abstract base classes, defining, 254
abstract class interface, 254
access control, 205-207
access specifiers, 205-207
accumulate function, 101
aggregate initialization, 203
algorithms
 averages, 99-103
 classic, 89
 for loops, C-style, 103-105
 item removal, 95
 iterators, 94-95
 minmax, 89
 predicates, 91-94
 unary, 92
 range algorithms, 89
 search algorithms, 91-94
aliases, 239
angle brackets
 template parameters, 63
 templates, 36

anonymous functions (see lambdas)
append file mode, 167
arguments, 13
 command-line, 179-180
 template argument deduction, 295
arithmetic mean, 99
arrays, 62
 C-style, 178
 decaying to pointer, 178
 class templates, 63
 elements, setting, 66
 inserting numbers, 66
arrow operator, 238, 239
ASCII characters, 11
assert macro, 24-26
assignment operators, 225
assignments
 copy, 225-228
 move, 225-228
associative containers, 59, 293
at function, 75
attributes, 31
auto keyword, 51
averages, 99-103

B

bad function, 22
base classes
 abstract, defining, 254
 derived classes, 254
binary operators, 106-107
binary predicates, 106-107
bitmasks, 170
bitwise operators, 169-170

- block scope, 9, 39
- bool operator, 22
 - returning, 28
- Boolean context, 23
- brace initialization, 15, 202
- braces, 202
 - (see also curly braces)
- break keyword, 60, 145
- buckets, 294
- buffer, flushing, 14
- build systems, 89

C

- C with classes, 201
- C-style arrays, 178
 - decaying to pointer, 178
- C-style for loops, 103-105
- call operators, 106, 137
- call stack, 54
- capacity function, 77
- capture groups, 123
- cassert header, 96
- catch block, 47-49
 - positioning, 54-55
 - std::exception, 53
 - std::invalid_argument, 53
 - walking the call stack, 54
- catch statement, 44
- catching exceptions, 44
- chaining, 13
- char type, 177
- character input, 16-18
- Clang, 4
 - source files, building, 7
 - versions, 6
- clang++, 6
- clashes, 294
- class keyword, 206
- class template argument deduction (CTAD), 63, 297
- class templates, 49, 295-306
 - arrays, 63
 - member functions, 298
 - vectors, 295
- classes
 - abstract
 - defining, 254
 - interfaces, 254
 - C with classes, 201

- derived, 254, 255-258, 264-271
 - polymorphism, 258-261
- Exchange, 259-264
- functions, 206
- hierarchy, 253, 254-258, 288
- classic algorithms, 89
- clear function, 35
- closed ranges, 95
- CMake, 89
- command-line arguments, 179-180
- comparison operators, 304
- Compiler Explorer, 3
- compilers, 1
 - CppReference, 6
 - warnings, 16
- const variable, 16, 204
- constants, 16
- constructors, 208-211, 215-217
 - copy, 222-223, 225, 232-233
 - delegating, 249
 - move, 223-224, 231-232
- containers
 - associative, 59, 293
 - std::map, 293
 - std::unordered_map, 293
 - initializer list, 77
 - initializing, 76-77
 - iteration, 74
 - range-based for loop, 68-71
 - sequenced, 59
 - sequential
 - double-ended queue, 80
 - std::array, 71
 - std::deque, 80
 - std::vector, 71-76, 211-214
- copy assignments, 225-228, 232-233
- copy constructors, 222-223, 225, 232-233
- Core Guidelines, 16
- .cpp file extension, 84
- CppReference, 22
- CTAD (class template argument deduction), 63, 297
- curly braces, 77
 - brace initialization, 202
 - for loops, 68
 - initializing vectors, 80
- custom deleters, 244-245

D

- dangling references, 127
- decaying to a pointer, 178
- declarations
 - functions, 84
 - header files, 84
- deep copy, 230
- delegating constructors, 249
- delete keyword, 237
- deleters, custom, 244-245
- dereference operators, 94
- dereferencing, 238
- derived classes, 254, 255-258, 264-271
 - polymorphism, 258-261
- destructors, 208, 210-211, 215-217
 - strings, 230
 - virtual, 271-273
- dictionaries (see lookup tables)
- discrete_distribution, 284
- distribution of random numbers, 136
 - discrete distribution, 284
 - Gaussian, 147-150
 - operators, 137
 - uniform, 136, 139-142
 - unsigned numbers, 146
- dot operators, 20
- double number, 53
- double value type, 38
- double-precision floating-point numbers, 18-19
- dynamic memory, 235
- dynamic polymorphism, 258

E

- empty vectors, 100
- encapsulation, 207
- engines
 - random-number engines, 137
 - seeds, 138
- eof (end of file) function, 20
- erroneous behavior, 33
- error handling, exceptions, 44
- errors, input, 34-37
- escape characters, 12
- Event function, 284-287, 302-304
- exceptions, 43
 - catching, 44
 - exception handling, 48
 - naming, 48
 - noexcept function, 43

- std::exception, 52
- std::invalid_argument, 52
- terminate, 47
- throwing, 44, 46-47
- try/catch block, 47-49
- uncaught, 55

- Exchange class, 259-264

- exists function, 172

- expectations, 49-51

- without value, 55-56

- expected values, 49

- explicit keyword, 23, 248

F

- fail function, 22

- file modes

- append, 167

- file-opening, 168

- file streams

- ifstream, 165-167

- input file streams, 165

- ofstream, 165-167

- output file streams, 159

- filenames, fully pathed, 163

- files

- bitwise operators, 169-170

- directories, 182

- opening, troubleshooting, 160

- paths, 182

- reading from, 165-167

- writing to

- output file streams, 159

- troubleshooting, 161-163

- filesystem library, 163-164

- exists function, 172

- filenames, fully pathed, 163

- find_if function, 121

- flags

- C++ version, 6

- Clang version, 6

- /W4, 6

- Wall, 6

- floating point numbers, 146

- double-precision, 18-19

- flushing, 14

- for loops, 99

- C-style, 103-105

- range-based, 68-71

- format strings, 188-190

- formatting, 187-188
- friend functions, 304
- friend keyword, 304
- function heads, 5
- function objects, 106
- function templates, 295-306
 - template parameter list, 295
- functions
 - accumulate, 101
 - arguments, 13
 - at, 75
 - attributes, 31
 - bad, 22
 - block scope, 9
 - body, 5
 - capacity, 77
 - classes, 206
 - clear, 35
 - curly braces, 5
 - declaring, 12, 84
 - defining, 12, 86
 - eof (end of file), 20
 - Event, 284-287, 302-304
 - exists, 172
 - fail, 22
 - file.close(), 160
 - find_if, 121
 - friend functions, 304
 - get_number, 26-29, 37-41
 - if, 22
 - ignore, 35
 - instance functions, 20
 - keywords, void, 4
 - main, 5
 - manipulators, 14
 - member functions (see member functions)
 - nodiscard, 31
 - noexcept, 43, 223
 - numeric input, 26-27
 - operands, 13
 - overloads, 12, 139-142
 - parameters, 5
 - print, 8
 - println, 8-10
 - push_back, 73
 - remove_if, 95-99
 - remove_invalid, 96-99
 - signature, 5
 - size, 66

- standard library, 8
- static member functions, 19
- static_assert, 251
- substr, 182, 184
- test functions, 24-26
- test_code, 25
- virtual, 255, 271-275

G

- g++, 6
- Gaussian distribution, 147-150
- GCC (GNU Compiler Collection), 3, 4, 6
- get_number function, 26-29
 - int, 37-41
- GNU Compiler Collection (GCC) (see GCC (GNU Compiler Collection))
- GNU Make, 89
- Godbolt, 3

H

- half-open ranges, 95
- hashes, 294
 - clashes, 294
 - std::hash, 299-302
- has_value, 50, 55, 56
- header files
 - benefits, 84
 - declarations, 84
 - function definitions, 217-218
- heap, 229
 - dynamic memory, 235
- Hello, world!, 7-14
- hierarchy of classes, 253, 254-258
 - std::variant and, 288
- .hpp file extension, 84

I

- IDEs (integrated development environments), 5
- if function, 22
- if statements
 - { } (curly braces), 60
- ifstream, 165-167
- ignore function, 35
- implementation defined values, 138, 169
- implicit conversion, 53
- include statement, 8
- increment operators
 - postincrement, 67

- preincrement, 66
- indexes, 63
- initial input code, 59
- initializer list, 77
 - aggregate initialization, 203
 - member initializer list, 209
- inline keyword, 91
- input
 - character input, 16-18
 - error clearing, 34-37
 - initial input code, 59
 - loops and, 59
 - numbers, floating-point, 18-19
 - with tests, 24-33
- input file streams, 165
- input streams
 - eof function, 20
 - std::cin, 15
- instance functions, 20
- instances, 203
- int (integer) value type, 4
 - get_number function, 37-41
- integrated development environments (IDEs), 5
- interfaces, classes, 254
- intermediate languages, 1
- International Organization for Standardization (ISO), 5
- interpreted languages, 1
- invalid pointers, 240
- invalidated references, 104
- iostream header, 11
- ISO (International Organization for Standardization), 5
- ISOCpp, 5
- iteration, 74
- iteration expressions, 103
- iterators, 73
 - algorithms, 94-95
 - begin, 95
 - end, 95
 - find_if function, 121
 - for loops, 104
 - half-open ranges, 95

K

- keywords
 - auto, 51
 - break, 60, 145
 - catch, 44

- char, 144
- class, 206
- concept, 311
- const, 16
- delete, 237
- double, 38
- explicit, 23, 248
- float, 146
- for, 99
- friend, 304
- if, 22
- inline, 91
- int, 4, 146
- long, 146
- mutable, 126
- noexcept, 223
- private, 205
- protected, 205
- public, 205
- short, 146
- static, 284
- struct, 202
- this, 232
- throw, 44
- try, 44
- typename, 295
- union, 277
- unsigned, 66
- virtual, 252, 254
- void, 4
- wchar_t, 195
- while, 60, 88

L

- lambdas, 109
 - behavior and, 111-117
 - calling, 111
 - captures, 120-123
 - by reference, 127-128
 - by value, 123-127
 - no-op, 165
 - variables, 110
- languages
 - intermediate, 1
 - interpreted, 1
- lazy evaluation, 132
- lazy instantiation, 298
- lazy views, 184
- linkers, 1

- Linux, 3-4
- literals, string literals, 178
- lookup tables, 293-295
 - key-value pairs, 293

- loops
 - counter increases, 66
 - for loops, 99, 103-105
 - input and, 59
 - range-based for loops, 68-71
 - raw loops, 105
 - while, 88
 - while loops, 60-62

M

- macOS Clang, 4
- macros, assert, 24-26
- main function, 5
 - calling new, 32-33
 - exception handling code, 48
- manipulators, 14, 126, 147
- member functions, 221
 - (see also special member functions)
 - const, 214
 - constructors, 208
 - copy assignment, 225-228
 - destructors, 208
 - move assignments, 225-228
 - noexcept, 223
 - output, 215-217
 - public, 206
 - rule of zero, 227
- member initializer list, 209, 211
- member variables
 - access, 203-207
 - member initializer list, 211
 - stock class, 211-214
- memory
 - dynamic, 235
 - heap, 229
 - pointers, invalid memory, 239
- memory leaks, 237
- Mersenne prime, 138
- Mersenne Twister, 138, 284
- metaprogramming, 251
- Microsoft Visual Studio, 4
- minmax algorithm, 89
- modules, 8
- move assignments, 225-228, 231-232
- move constructor, 223-224, 231-232

- move semantics, 221, 226
- moved-from objects, 224
- mutable keyword, 126

N

- namespaces, 9, 86-89
 - using namespace, 191
- negative numbers, 53
 - ranges view, 117-120
 - removing, 109-117
- nested replacement field, 193
 - (see also format strings)
- newline characters, 9, 13
 - appending, 13
- no-op lambdas, 165
- nodiscard, 31
- noexcept function, 43
- noexcept keyword, 223
- nontype template parameters (NTTP), 63, 313
- normal distribution, 147
 - (see also Gaussian distribution)
- NTTP (nontype template parameters), 313
- null character, strings, 178
- numbers
 - adding to vectors, 75
 - floating point, 18-19, 146
 - int, 4, 146
 - long, 146
 - negative
 - ranges view, 117-120
 - removing, 109-117
 - pseudorandom, 135
 - real numbers, 146
 - unsigned, 66
 - whole, 146
- numeric input, 17
 - fail function, 22
 - functions, 26-27
 - signatures, 24
 - streams, 27-29

O

- object types, 202
- object-oriented programming (OOP) (see OOP (object-oriented programming))
- objects
 - classes, 33
 - copying, 221-223
 - assignment operators, 225-228

- function objects, 106
- hashes, 294
- lifetime, 282
- moved-from, 224
- moving, 223-224
 - assignment operators, 225-228
- scope, 282
- temporary, 195
 - rvalues, 223-224
- ODR (one-definition rule), 91
- ofstream, 165-167
- one-definition rule (ODR), 91
- OOP (object-oriented programming), 253
- open ranges, 95
- operands, 13
- operators, 13
 - &&, 20
 - >, 238
 - ::, 19
 - <<, 13
 - ==, 303
 - >>, 16, 30
 - arrow, 238, 239
 - assignment, 225
 - binary, 106-107
 - bitwise, 169-170
 - bool, 22, 28
 - call operators, 106, 137
 - comparison, 304
 - dereference, 94
 - dot, 20
 - incrementing, 66
 - postincrement, 67
 - preincrement, 66
 - random number distribution, 137
 - ~ (tilde), 170
- output file streams, 159, 165-167
- overloaded functions, 12
- overloads, 139-142

P

- parameters, 5
 - member variables, 208
 - nontype template parameters, 63
 - NTTP (nontype template parameters), 313
 - passing by reference, 26
 - passing by value, 26
 - templates, 63
- passing by reference, 26

- passing by value, 26
- pointers
 - dereferencing, 238
 - invalid, 240
 - invalid memory, 239
 - nullptr, 238
 - raw, 239-252
 - smart, 239-252
- polymorphism
 - dynamic polymorphism, 258
 - static polymorphism, 258
- postincrement operator, 67
- pragma directive, 85
- predicates in algorithms, 91-94
 - binary, 106-107
 - unary, 92
- preincrement operator, 66
- preprocessing, 85
- print header, 8
- println function, 8-10
- private access specifier, 205-207
- private keyword, 205
- protected keyword, 205
- pseudorandom numbers, 135
 - (see also random numbers)
- public access specifier, 205-207
- public keyword, 205
- public member functions, 206
- push_back function, 73

Q

- qualifiers, 214

R

- RAII (Resource Acquisition Is Initialization), 237
- random library, 136
- random numbers
 - distribution, 136
 - Gaussian, 147-150
 - operators, 137
 - uniform, 136, 139-142
 - unsigned numbers, 146
 - weighted, 136
 - engines, 137
 - generating, 136-139
 - pseudorandom, 135
 - seeds, 150-154
- random_device, 150

- range adapters, 128
- range algorithms, 89
- range views
 - composing, 128-131
 - lazy, 132-133
- range-based for loops, 68-71
- ranges, 89
 - closed, 95
 - half-open, 95
 - open, 95
- ranges view
 - chaining, 117
 - filter function, 118
 - negative numbers, 117-120
- raw loops, 105
- raw pointers, 239-252
- reading from files, 165-167
- real numbers, 146
- refactoring, 31-32
- reference semantics, 287
- references
 - aliases, 239
 - dangling, 127
 - invalidated, 104
 - lambda captured by, 127-128
 - passing by, 26
- remove_if function, 95-99
- replacement fields, 187
 - nested, 193
- Resource Acquisition Is Initialization (RAII), 237
- rule of five, 228
- rule of zero, 227
- rvalues, 223-224

S

- scientific notation, 21
- scope, 9, 282
 - block scope, 39
- scope-resolution operator, 9
- search algorithms, 91-94
- seeding engines, 138, 150-154
- segmentation fault, 56
- self-assignment, 232
- semantic models
 - reference semantics, 287
 - value semantics, 287
- semicolons in statements, 9
- sequenced containers, 59
- sequential containers
 - double-ended queue, 80
 - initializer list, 76-77
 - std::deque, 80
 - vectors
 - adding, 77-78
 - deleting from, 78-79
 - fixed values, 79-80
- short keyword, 146
- signal, 56
- signatures, 5
 - numeric input, 24
- SIGSEGV, 56
- size function, 66
- slicing, 273-275
- small string optimization, 229
- smart pointers, 235, 239-252
- source files
 - .cpp, 84
 - declarations in, 84
 - function definitions, 217-218
 - header files, 84-85
- special member functions
 - copy assignment, 225-228
 - copy assignments, 232-233
 - default constructors, 222
 - move assignments, 225-228
 - move constructor, 223-224
- specializing templates, 298-299
- stack unwinding, 54
- standard deviation, 147
- standard library, 8
- static keyword, 284
- static member functions, 19
- static polymorphism, 258
- static_assert function, 251
- std namespace, 9
- std::any, 289-291
- std::array, 62, 66
- std::bad_expected_access, 56
- std::cin, 15, 33
- std::cout, 11-12
- std::deque, 80
- std::endl, 13
- std::exception, 52, 53
- std::expected, 43, 49-51
- std::format, 188-190
- std::function, 112-114
- std::hash, 299-302

- std::invalid_argument, 52
- std::istream, 28
- std::map, 293
- std::numeric_limits, 86
- std::optional, 289-291
- std::print, 196
- std::println, 187-188, 196
- std::random_device, 138
- std::shared_ptr, 243-244
- std::sort, 106
- std::streamsize, 36
- std::string, 180-182, 195-198, 229-231
- std::stringstream, 27-29
- std::unique_ptr, 236-239
- std::unordered_map, 293-313
- std::variant, 278-283, 287-290
- std::vector, 71-76, 211-214
- std::visit, 281-283
- std::weak_ptr, 243-244
- stock class, 211-214
- stream extraction operator (>>), 16
- stream insertion operator (<<), 11
- streams, 11
 - clear function, 35
 - ifstream, 165-167
 - ignore function, 35
 - input file streams, 165
 - manipulators, 126
 - ofstream, 165-167
 - output file streams, 159
 - std::istream, 28
 - std::stringstream, 27-29
- string literals, 178
 - characters, 178
- string views, 184-186
- strings, 195
 - copying, 230
 - destructors, 230
 - format strings, 188-190
 - null character, 178
 - small string optimization, 229
- string_view, 177
- strongly typed languages, 38
- struct, 204
 - declaring, 202
- structured bindings, 310
- substr function, 182, 184
- syntactic sugar, 22

T

- tables
 - lookup tables, 293-295
 - virtual function tables, 273-275
- TDD (test-driven development), 24
- template argument deduction, 295
- template parameter list, 295
- templates
 - angle brackets, 36
 - arrays, 63
 - class templates, 49, 295-306
 - function templates, 295-306
 - NTTP (nontype template parameters), 313
 - parameters, 63
 - nontype, 63
 - random library, 136
 - specializing, 298-299
- temporary objects, 195
 - rvalues, 223-224
- test functions, 24-26
- test-driven development (TDD), 24
- test_code function, 25
- this keyword, 232
- throwing exceptions, 44, 46-47
- tool installation, 3-4
- trading game, 143-145, 190-194
 - Event function, 284-287
 - event tally, 306-311
 - Exchange class, 261-264
- translation unit, 91
- troubleshooting
 - compiles, 10
 - std::variant and, 288
 - writing to files, 161-163
- try statement, 44
- try/catch block, 47-49
- typename keyword, 295
- types
 - properties, 250

U

- unary predicates, 92
- uncaught exceptions, 55
- undefined behavior, 33
- unexpected values, 49
- uniform distribution, 136, 139-142
- uninitialized variables
 - erroneous behavior, 33
 - undefined behavior, 33

- union keyword, 277
- unsigned numbers, 66, 67
- unused variables, 16
- UTF8, 195

V

- value semantics, 287
- value types
 - double, 38
 - expected, 49
 - int (integer), 4
 - strongly typed languages, 38
 - unexpected, 49
- values
 - displaying, 68-71
 - implementation defined, 169
 - lambda captures, 123-127
 - passing by, 26
 - vectors, 154-156
- variables
 - brace initialization, 15
 - const, 16
 - constants, 16
 - declaring, 15-16
 - initializing, 33
 - lambdas, 110
 - member
 - parameters, 208
 - member, access, 203-207
 - uninitialized
 - erroneous behavior, 33
 - undefined behavior, 33
 - unused, 16
 - values, 16
 - visibility, 41
- vectors

- adding numbers, 75
- class templates, 295
- declaring, 140
- deleting elements, 78-79
- elements, adding, 72-74, 77-78
- empty, 100
- initializing, fixed values, 79
- values, 154-156
- views
 - lazy, 184
 - string, 184-186
 - string_view, 177
- virtual destructors, 271-273
- virtual function table, 273-275
- virtual functions, 255, 271-273
 - slicing, 273-275
 - virtual destructors, 271-273
- virtual keyword, 252, 254
- Visual Studio, 4
- volatility, finance, 149

W

- /W4, 6
- walking the call stack, 54
- Wall, 6
- warnings, 6, 16
- wchar_t keyword, 195
- weighted distribution, 136
- WG21, 5
- while loops, 60-62, 88
- whole numbers, 146
- wide characters, 195
- Windows
 - Microsoft compiler, 4
 - source files, 7

About the Author

Frances Buontempo is the editor of ACCU's *Overload* magazine, which has a focus on C++. She has published articles and given talks centered on technology and machine learning. With a PhD in data mining, she has been programming professionally since the 1990s. During her career as a programmer, she has championed unit testing, mentored newer developers, deleted quite a bit of code, and fixed a variety of bugs. She has experience teaching and training and can make complicated subjects understandable.

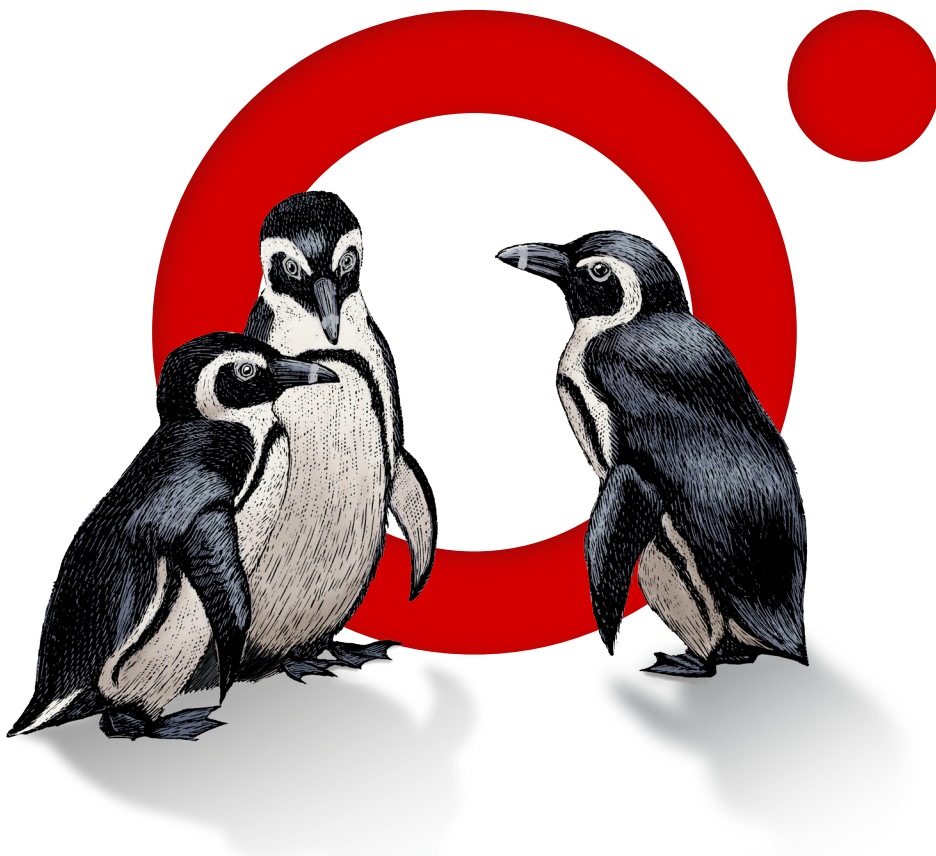
Colophon

The animal on the cover of *Introducing C++* is the eclectus parrot (*Eclectus roratus*). The eclectus parrot can be found throughout rainforests in the Solomon Islands, New Guinea, Australia, and Indonesia. Not only are they striking in color, but they are among the most talented of talking parrots, capable of developing an extensive vocabulary and mimicking various sounds.

This species is known for its sexual dimorphism—unusually, the females are more brightly colored than males. This dimorphism is so distinctive that until the 20th century, males and females were mistakenly classified as separate species. Females boast deep violet-blue and red feathers, complemented by a dark beak. Males are primarily emerald green with hints of blue and red underneath their wings and a vivid orange and yellow beak.

Although eclectus parrots display reverse sexual dimorphism in terms of their coloration, they do not exhibit the behavioral role reversal usually associated with this characteristic. The emerald green color of the male eclectus parrot allows it to blend in the rainforest while it forages for food while females remain near the nest. They typically nest in high trees and reuse one nest for their entire lives since finding suitable locations is challenging. Because of this, it's common for females to defend their nest against intrusion from other females.

The cover illustration is by José Marzan Jr. based on an antique line engraving from Lydekker's *Royal Natural History*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

**Learn from experts.
Become one yourself.**

60,000+ titles | Live events with experts | Role-based courses
Interactive learning | Certification preparation | Verifiable skills

Try the O'Reilly learning platform free for 10 days.



"Finally, a C++ book that saves the sharp edges for later and lets you build real things first. Modern, practical, and long overdue."

Matt Godbolt, Compiler Explorer

"It's wonderful to see a fresh book that not only uses C++ to teach programming from scratch but that starts with modern C++23! With the number of C++ programmers worldwide growing every year with no sign of slowing down, this book is very timely."

Herb Sutter, ISO C++ committee chair

Introducing C++

You know how to code, but you're ready to level up. You've heard about the power and performance of C++, a language vital to fields like AI, game development, and high-performance computing. But where do you even start? Existing resources are outdated or overly academic, or they assume a deep understanding of C. You need a guide that respects your existing skills and gets you coding fast with modern C++.

This book cuts through the noise, focusing on the essential elements of C++ with hands-on projects that quickly build your skills and confidence. Author and C++ expert Frances Buontempo gets you up to speed quickly with the latest features and best practices of the language, preparing you for more advanced exploration.

- Write clean and efficient C++ code
- Understand core concepts and syntax
- Apply various programming approaches, from OOP to functional styles
- Read and understand complex C++ definitions and resources

Frances Buontempo is the editor of ACCU's magazine, *Overload*, and has been a professional programmer since the 1990s. She holds a PhD in data mining and writes and speaks about C++ and machine learning. She mentors developers, promotes unit testing, and makes complex topics easy to understand.

PROGRAMMING / C++

US \$59.99 CAN \$74.99

ISBN: 978-1-098-17814-7

